FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

OPC UA support for Beremiz softPLC

Martim Afonso Maia Henriques da Silva

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Mário Jorge Rodrigues de Sousa

July 28, 2018

Resumo

Integração do protocolo de comunicação para Automação Industrial OPC UA no softPLC Beremiz, o qual segue a norma IEC 61131-3. Mapeamento dos elementos definidos no IEC 61131-3 para OPC UA utilizando como guia a especificação criada pela OPC Foundation em conjunto com a fundação PLCopen "OPC UA Information Model for IEC 61131-3".

Abstract

Integration of the Industrial Automation protocol OPC UA in the IEC 61131-3 compliant Beremiz softPLC. The mapping between IEC 61131-3 elements and OPC UA will follow the "OPC UA Information Model for IEC 61131-3" specification.

Contents

List of Figures

List of Tables

Listings

Abbreviations & Symbols

A&E	Alarms & Events
API	Application Programming Interface
CAN	Controller Area Network
COM	Component Object Model
DA	Data Access
DCOM	Distributed Component Object Model
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDA	Historical Data Access
HMI	Human-Machine Interface
IEC	International Electro-technical Commission
IoT	Internet of Things
IIoT	Industrial Internet of Things
IoTS	Internet of Things and Services
M2M	Machine to Machine
MES	Manufacturing Execution System
OLE	Object Linking & Embedding
PC	Personal Computer
PLC	Programmable Logic Controller
SCADA	Supervisory Control and Data Acquisition
SOA	Service-Oriented Architecture
TC	Technical Committee
TR	Technical Report
UA	Unified Architecture
XML	Extensible Markup Language

Chapter 1

Introduction

We are on the brink of a new Age. The Age of a fully connected world, where the portable devices we keep in our pockets are now portals into the *digital matrix*, connecting us to an all encompassing network. This network is a digital living system, with packets of bits flowing through the wires and the waves, and is expanding at an exponential rate. The immense flow and sharing of information is disrupting the way we communicate, how we socialize, how we learn, the way enterprises develop their business and how industries operate.

Figure 1.1: Digital World [1]

This new vision is what it's now called the Fourth Industrial Revolution. Klaus Schwab, Founder and Executive Chairman of the World Economic Forum, in his book - *The Fourth Industrial Revolution* [18] - explains how new technologies and breakthroughs in robotics, artificial intelligence, nanotechnology, renewable energy, quantum computing, biology and others are blurring the lines of the physical, biological and digital realms. The sheer velocity, scope and impact

of this new advances in technology is transforming the landscape of all sectors of Industry and Society and bringing humanity into the Fourth Industrial Revolution.

In the First Industrial Revolution, the energy revolution brought by the spread of steam power engines, enabled the mechanization of labor, leading us into a new Age of factories and mechanical production. In the Second Industrial Revolution, the advent of electricity lead us into a new energy revolution, and with the electrification of factories and division of labour, the assembly line began being implemented in factories to produce large scales of standardized product and bring us into the Age of Mass Production. The Third Industrial Revolution, also called Computer or Digital Revolution, is characterized by developments such as semiconductors, personal computing and the internet. In the Industry sector, the introduction of digital computers systems - Programmable Logic Controllers - in the factory floor brought an increase automation of production and the beginning of Automation Programming jobs - 3.1 PLC History. And the use of digital computers to store, analyze and retrieve data spread through the Enterprise Sector to give birth to the Information Technology field.

We are on the first steps of this new Industrial Revolution, falling prices of hardware are enabling the exponential growth of the number of devices connected to the Internet, Every "Thing", from wind generators, cars and buildings to information systems and people can now share data with each other across the *Internet of Things* (IoT). Sectors such as Transportation, Health, Cities, Energy Distribution or Manufacturing are embracing the new degree of sharing and connectivity, and earning the "Smart" nickname. In the context of Manufacturing the concept of "Smart Factory" is being created, based on the bridging of production methods with information technologies and communication systems.

Various platforms comprising Industrial Manufacturers and Information and Communication Technology Companies are being created to bring forward the concepts of "Smart Factory" and the connection between the various industries sectors. In Germany, Plattform Industrie 4.0, was created to led Germany's Manufacturing Industry into the Fourth Industrial Revolution, and is developing state-of-the-art Architecture Models, focused on the horizontal connection of industrial assetss, for the development of the "Smart Factory" and their integration into Digital Supply Chains. In the United States, the Industrial Internet Consortium (IIC), is a platform of leading Automation Manufacturers and Information Technology Companies, that is also developing leading Architecture Models for their vision of the Industrial Internet of Things (IIoT), a more vertical integration oriented vision, with the objective of connecting the various Industries Sectors together into the IIoT. 2.2.4 Manufacturing in the Fourth Industrial Revolution - *Industry 4.0*

OPC Unified Architecture (OPC UA) is a recognized standard in industrial automation addressing interoperability and data exchange from the factory floor level, integrating sensors, actuators and controllers, to central servers hosting enterprise applications, being them in the enterprise private domain or in public clouds. It is being promoted by this various platforms as the chosen standard to connect the various industrial devices - such as the Programmable Logic Controller - horizontally, providing communication between devices, in the same Factory Hierarchy Level,

and vertical integration between devices and applications, across the various Levels of the Factory Hierarchy.

Beremiz - 4.1 Beremiz IDE - is an open source Integrated Development Environment for programming Programmable Logic Controllers according to the IEC 61131-3 - a standard on how to program PLCs 3.3 IEC 61131-3 - with a software PLC used for run-time simulation and testing of the programs developed, that can be run in any PC architecture, enabling students to have a free framework for practicing PLC programming.

1.1 Objectives

The raising relevance of OPC UA in automation industry along with the success in the adoption of standard industrial development environments created within a open source software development model lead us to the challenge of upgrading Beremiz so that it can support the OPC UA standard. As physical PLCs in the factory floor are now embedded with OPC UA Servers to enable exchange of process data and commands horizontally and vertically, the Beremiz software PLC will be embedded with it's own OPC UA Server. This will enable students, to practice programming PLCs according to the IEC 61131-3 standard and connect them with developed Industrial software Applications using this Industrial Standard. Preparing Students to real-world Automation scenarios in the "Smart Factory" of the future.

1.2 Structure

This dissertation is structured in eight chapters including the Introduction. The State of the Art is divided into three parts. The first one introduces the OPC UA communication standard and it's predecessor OPC Classic. The second introduces the IEC 61131-3 programming standard and the third and last part of the State of the Art introduces the Beremiz project and the MatPLC IEC Compiler, both designed as IEC 61131-3 compliant software tools.

The fifth chapter describes the specifications used during this project, the sixth chapter details the Software library used and the seventh chapter describes the work developed to provide the OPC UA support.

The last chapter deals with the Conclusions and Future Work. Further details on the project are available online at https://ee12099.github.io.

Chapter 2

State of the Art I - OPC UA

In this chapter the OPC UA communication protocol, standardized as IEC 62541, and it's predecessor, the OPC Classic industry standard, are described.

In the first part of this chapter the OPC Classic is described as a standard adopted by Industry to solve the Integration problems associated with the proprietary custom drivers used to provide data exchange between control devices and PC applications.

The second part of this chapter introduces the cross-platform and web-friendly version of OPC Classic, the OPC Unified Architecture (UA). This new extensible standard, designed using a Service-Oriented Architecture, and with an Object-Oriented representation of data and information, is taking Industries by storm.

2.1 OPC Classic

2.1.1 Brewing the technologies

The progressive introduction of computerized systems in the control of industrial processes, since late 70ths - as powerful centralized processing stations or as distributed control nodes - has brought the need to overcome communication challenges, in the flowing of commands and collecting data, within time, security or reliability constraints.

With the progress and emergence of computerized solutions from different manufacturers on the shop floor, the complexity has increased particularly in what concerns the inherent problems of interoperability costs and in dependence from sole manufacturers (vendor lock-in).

Those were challenges common to industries adopting computers and Information Technology. Hopefully IT companies, as Microsoft, were continuously investing and advancing the state of the art, namely in delivery of technical solutions in the area of inter-process communications, the basis for technical construction of communications standards specific for the industrial environment.

DDE (Dynamic Data Exchange) was the predecessor to OLE. It was introduced with Windows 2.0 in 1987. DDE is a standard mechanism for inter-process communication that enables one application to exchange data with another application at run-time. [19]

In 1990 Windows 3.0 was launched and with it the brand new DDE evolution - OLE (Object Linking and Embedding) 1.0. While DDE was able to transfer limited amounts of data between running applications, OLE was able to maintain active links between two documents and even embedding one type of document within another. OLE 1.0 would evolve into a fully fledged architecture for software components - the Component Object Model (COM) and it's network version Distributed COM (DCOM) - and be reimplemented with this new approach to become OLE 2.0 in 1992.

2.1.2 Forming the Task Force

In that same year, a task group comprised of members from the industrial control and data acquisition areas began meeting at Microsoft. It was called WinSEM (Windows in Science, Engineering and Manufacturing) and by 1994, through WinSEM work, the idea of adopting OLE technologies has the way to implement real-time communication within industrial environments became to emerge. [19]

SCADA vendors became interested in standardizing the interface between their SCADA core and the device drivers. This way vendor's would only need to maintain a driver that would work with all Windows software. [19]

With this vision in mind, a task force of automation vendors, comprised of Fisher-Rosemount, Intellution, Opto 22, Intuitive Technology and Rockwell Software, was formed with the mission of developing a standard for data access based on Microsoft's COM and DCOM. [3] [19] [20] This was called OPC, an abbreviation for OLE for Process Control.

The OPC Task Force, branded as OPC Foundation, went public at the 1995 ISA Show in New Orleans. [19] [3]

2.1.3 Architecture

2.1.3.1 Before OPC - The Custom Driver Wild West

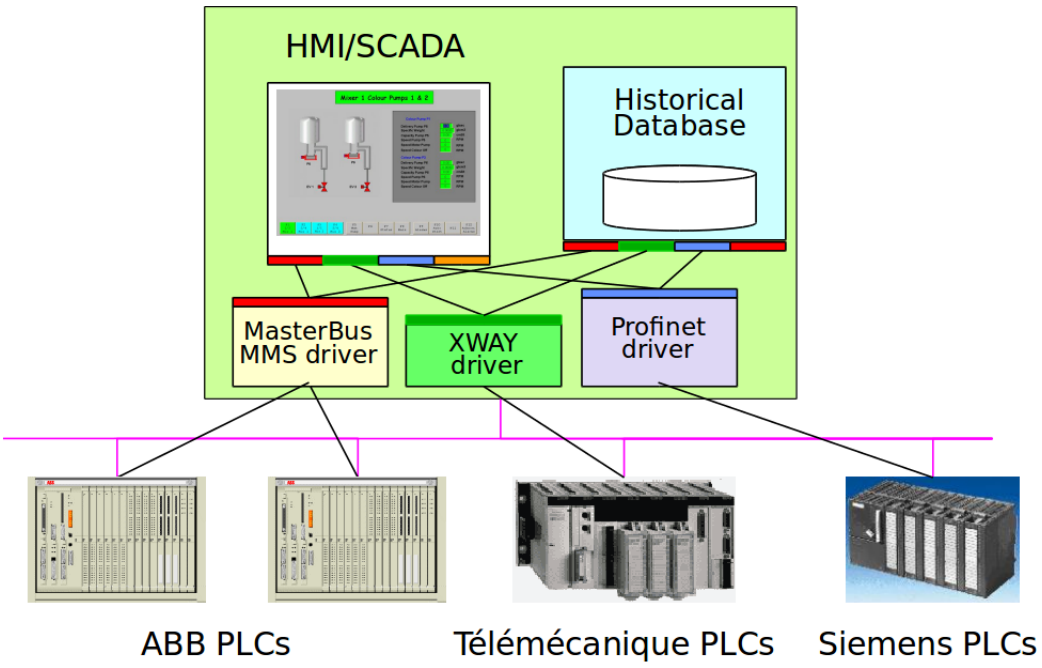

Figure 2.1: Conventional communication architecture [2]

In a conventional communication architecture, industrial software applications such as HMI, SCADA or databases, access data from control devices using custom device-specific drivers.[21] [22] This implies that the application level has to implement a specific communication stack for interfacing with each proprietary driver. [20]

If this type of scenario is manageable with a few types of equipment's, the growing of the Industrial Automation market, raises a lot of challenges:

- When a new device from a different vendor is added to the plant-floor, for all the preexisting applications to be able to access data from this new device, a custom-communication-driver must be developed and added to each application stack. This leads to various drivers per application that need to be developed and then maintained.[22] [21] [20]
- Because the software application needs to include the driver for a different vendor device, and this driver needs to be maintained, the cost of using a different vendor's product is too high. This creates the so called vendor-lock effect. [23]
- Changes in the devices hardware's capabilities, if not accompanied by upgrades in the drivers may cause functionality problems. [21] [22]

- When two packages containing independent drivers try to access the same device simultaneously, access conflicts may occur. [21] [22]

The hardware manufacturers tried to develop and maintain their own software application drivers but the multitude of client protocols made that task nearly impossible. [21]

2.1.3.2 OPC - The Middleman brings Law and Order

This growing complexity and cost inefficiency led the Industry to work together in the standardization of key Industrial Automation Interfaces, namely the interface between control applications and field/control devices.

The result of this standardizing initiative was the creation of OPC. OPC was based on client-server architecture [20] and would work as a "middle-man" that would convert generic read/write requests into device-specific requests and the other way around. [24]

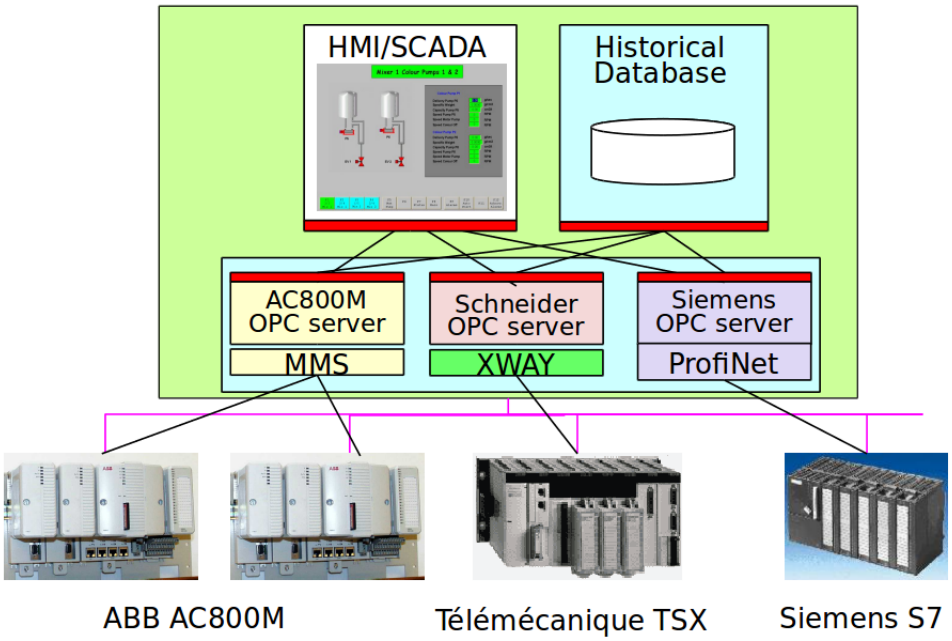

Figure 2.2: OPC based communication architecture [2]

The OPC specification draw a much needed line between hardware providers and software developers. [21]. With this new interface automation vendor's just need to focus on creating quality devices and providing a communication driver for the OPC Server. Software applications began to use the same OPC COM client interface enabling a plug-and-play concept of interoperability for the market dominant Office products and Windows based industrial software applications, like MES and ERP. [21].

2.1.4 OPC Classic Specifications

2.1.4.1 OPC Data Access (OPC DA) Specification

OPC Data Access was the first OPC Classic specification. The first simplified version was released in August 1996. [3] The corrected version 1.0A was released in 1997 and was the first adopted by the market. [19] This specification proposed the client-server architecture for real-time process data exchange. [20].

Variable	Attribute
Process Value	*Value*
Timestamp	*Time/Date*
Quality	*Good/Unknown/Bad*

Each server variable contains three properties: *Value*, *Quality* and *Timestamp*. [20] The OPC DA server reads the process value of a given variable from the field device and stores it in that variables *Value* property. Besides the *Value* the server set's the *Quality* property that represents how well the stored *Value* matches the device actual value, and a *Timestamp* from when this two other properties where set. [25]

2.1.4.2 OPC Alarms & Events (OPC A&E) Specification

This specification was released in 1999 and defined a common interface for the exchange of events and alarm conditions [26]. This new interface works only with event data, so the OPC A&E Server will work along side the OPC DA server. This way industrial end-users will have a common interface for their real-time data (DA) and another one for event data (A&E). [20]

2.1.4.3 OPC Historical Data Access (OPC HDA) Specification

This specification is somewhat an extension of the DA specification. It let's clients access raw data (historical data stored) and aggregated data (data extracted from the raw historical data and processed). Simple HDA servers only provide raw data, the more complex ones can expose aggregated data, process it and provide insights such as average values, trends, annotations, and so on. This historical data can provide valuable insight in process optimization and quality evaluation. [20] [27]

2.1.4.4 OPC Batch Specification

This specification is an extension of the DA specification for the special case of batch processes. [20] Batch processing manufacturing was standardized in the IEC 61512-1 norm. This norm specifies visualization, report generation, sequence control systems and equipment. This specification was designed to provide interoperability to all this software components. [20]

2.1.4.5 OPC Data eXchange (OPC DX) Specification

This specification extended the DA specification for horizontal server-to-server communication. [20] It can be used for the case of an implementation with a back-up or redundant sever strategy.

2.1.4.6 OPC Commands Specification

This specification was conceived to fill a functional gap on OPC specifications, namely, the capability to remotely initiate commands to be executed, to configure a device or system, or to start a program reload. [20]

2.1.4.7 OPC XML-DA Specification

This specification was an attempt to provide platform independence by porting OPC specifications from the COM/DCOM technologies to Web services. Because OPC DA was based on COM/DCOM it was only successfully implemented in Windows platform. Besides that, DCOM technologies were not firewall friendly. To overcome these problems OPC Foundation created this standard, replacing COM/DCOM for HTTP/SOAP and Web Service technologies. However this portability came at a price, the XML transmission is less efficient than the binary data transmission of the DCOM technology. [27]

As explained in [20], this standard was somewhat of a predecessor to OPC UA:

"This standard can be seen as the predecessor of OPC UA, but were released too late. Customer, manufacturers and developers were waiting for OPC UA and not keen in developing a client-server architecture that would be soon overtaken by the new OPC UA specification."

2.1.4.8 OPC Limitations

In spite of the success of the OPC, materialized in the good acceptance it received from the industries with its application in many automaton solutions, a series of limitations have been emerging over the years.

Some of these limitations result from the intrinsic design of OPC specifications. This is the case of the complexity of managing different OPC services, namely due to the fact that each specification (DA, AE or HDA) operates in a different address space, there was no connection between an actual value read with DA to the history read with HDA or the events raised based on the same value. [20]

Also the tight coupling with Microsoft's COM/DCOM technology that shaped OPC raises difficulties when there is the intention to operate OPC on other operating systems, namely on Linux that has been established as a credible alternative to Windows. The problem comes from the need to emulate COM/DCOM functionality in order to use it over Linux which has caused many problems. [27] [20]

Other weak area for OPC was security. Security concerns were not sufficiently valued at the mid 1990s. DCOM did have some firewall configuration issues [27] and security was mainly left to the responsibility of the operating system. [20]

Microsoft operating systems dominate the industrial automation landscape. Automation vendors begin using Microsoft's COM and DCOM in their product offerings.	1990s	
	1995	Automation vendors Fisher-Rosemount, Intellution, Opto 22 and Rockwell Software form a task force to develop a standard for data access based on COM and DCOM, and call it OPC, an abbreviation for OLE (Microsoft Object Linking & Embedding) for Process Control.
The task force, established a year earlier, releases version 1.0 of a simplified OPC specification for Data Access (DA) in August.Within the first year, several other software and hardware vendors began using OPC as their mechanism for interoperability.It soon becomes clear that a more formalized organization of compliance, interoperability, certification and validation is necessary. The OPC Foundation is established at the Chicago ISA Show in September.	1996	
	1998	The OPC Foundation begins converting its existing specification to web services.
OPC Alarms & Events (OPC AE) specification is released.	1999	
	2001	OPC Historical Data Access (OPC HDA), Batch and Security specification are released.
OPC Complex Data, Data eXchange and XML-DA specifications are released.OPC Unified Architecture (OPC UA), comprising of 13 separate parts, is created by the OPC Foundation.The original OPC specification is now referred to as "Classic OPC" or OPC Classic.	2003	
	2004	OPC Commands specification is released.
OPC UA version 1.0 becomes available.	2006	
	2007	OPC Certification Program and Test Labs are introduced.Automation vendors begin offering the first products based on OPC UA.
OPC UA version 1.01 becomes available.OPC UA for Analyzer Devices (ADI) is released as a companion specification driven by the Pharma and Chemical manufacturing industries.	2009	
	2010	The first embedded OPC UA devices are released.OPC UA for IEC 61131 is released as a companion specification.
IEC 62541 is released (UA).	2012	
	2013	OPC UA 1.02 is released.OPC UA for ISA-95 is released.The OPC Foundation supports over 480 members across China, Europe, Japan and North America.

Figure 2.3: OPC History [3]

2.2 OPC UA

Motivated by the limitations presented by the OPC and taking into consideration the advances in the state-of-the-art of information technology and in particular the emergence of service-oriented software architectures (SOA) and technologies as Web Services, OPC Foundation has decided to redesign its specifications, which in fact had been created almost 10 years ago, and in 2008 released the first version of the OPC *Unified Architecture* (UA) - a platform independent service-oriented architecture that integrates the functionality of the OPC Classic specification into one modular framework. [28] [29] The Classic OPC was accepted in the industry as a *de facto* standard but was never approved as a standard by IEC. For the OPC UA, the OPC Foundation

collaborated with IEC and since 2010 the OPC UA specifications are being standardized as the IEC 62541 standard.

2.2.1 Specifications

The latest OPC UA version (release 1.04) is structured in 14 parts, that can be organized in 3 groups. Parts 1 through 7 and Part 14 specify the core capabilities of OPC UA. These core capabilities define how data is modelled and exposed and the services used to access and manage it. Part 8 through 11 apply these core capabilities and data models to specific types for accessing different types of data. The OPC UA Part 12 describes Discovery mechanisms and Part 13 ways of aggregating data. OPC UA specifies the abstract set of services in Part 4, a security model in OPC UA Part 2, data structures in Part 5, network protocol mappings in OPC UA Part 6 and profiles for compliance in Part 7.

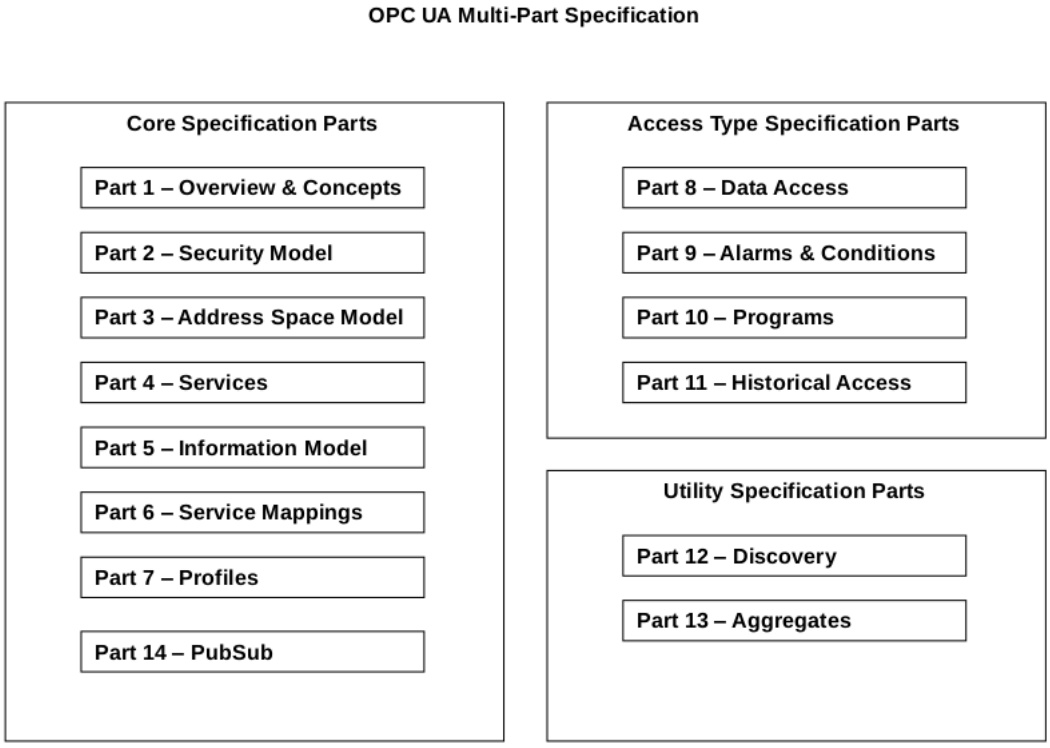

Figure 2.4: OPC UA Specifications [4]

"OPC UA defines generic services and in doing so follows the design paradigm of service-oriented architecture (SOA), with which a service provider receives requests, processes them and sends the results back". [4] A communication stack is used on client and server-side to encode and decode message requests and responses. Different communication stacks can work together as long as they use the same technology mapping. An overview of the UA stack is displayed in **??**. The interface between an OPC UA application and the Stack is a non-normative API which hides

the details of the Stack implementation. [30] Only the message formats for data exchange on the wire are specified. The API used by the UA Application depends on the development platform technology. OPC UA currently defines two mappings: UA Native, using a binary protocol, and Web Services. The encoding data can be done using 3 options: OPC UA Binary, OPC UA XML and OPC UA JSON. In addition, several protocols are defined: OPC UA TCP, HTTPS and WebSockets. The UA Binary was specified to provide low bandwidth data transfers [30] for embedded devices and other plant-floor systems where the use of XML Web Services would consume too much resources [20]. To provide compatibility and interoperability between both mappings, the UA Binary was designed based on Web Services logic. It mimics WS protocols in message structure and encryption, just the encoding is different. [30] With this communication stack the OPC UA provides an interoperability layer independent from any specific operating system or protocol. [20]

2.2.2 Services

- **Discovery Service Set**

  Defines *Services* to allow *Clients* to discover the *Server's* Endpoints and retrieve the security configuration of each Endpoint. Defined in OPC UA Part 4, section 5.4.

- **SecureChannel Service Set** (Part of the Communication Stack)

  Determines the security configuration of a server and establishes a secure communication channel between the client and the server. Defined in OPC UA Part 4, section 5.5.

- **Session Service Set** (Part of the Communication Stack)

  Establishes an application-layer connection in the context of a *Session* on behalf of a user. The API used by the UA Application depends on the development platform technology. Defined in OPC UA Part 4, section 5.6.

- **NodeManagement Service Set**

  Provides an interface for configuring UA servers. It allows clients to add, modify or delete nodes in the address space. Defined in OPC UA Part 4, section 5.7.

- **View Service Set**

  Enables clients to explore the structure of the address space by browsing nodes, navigate the hierarchy and follow references. Defined in OPC UA Part 4, section 5.8.

- **Query Service Set**

  Provides a way for a client to filter nodes from the address space based on a specified criteria. Defined in OPC UA Part 4, section 5.9.

- **Attribute Service Set**

Defines services that enable *Clients* to read and write Node's Attributes. Defined in OPC UA Part 4, section 5.10.

- **Method Service Set**

  Provides an interface for invoking the methods attached to an object. Defined in OPC UA Part 4, section 5.11.

- **MonitoredItem Service Set**

  Determines which attributes or events should be monitored for changes by a client. Defined in OPC UA Part 4, section 5.12.

- **Subscription Service Set**

  Used to generate, modify or delete messages for MonitoredItems. Also provides recovery of missed Messages and communication failures. Defined in OPC UA Part 4, section 5.13.

2.2.3 Data Model

With OPC UA, services for accessing specific types of data, such as Data Access (DA), Alarms & Events (A&E), Historical Data Access (HDA) and OPC Commands - previously exposed by separate OPC COM Servers - can now be accessed from a single OPC UA Server. Real world entities are represented as Objects with attached Variables, Events and Methods - Figure 2.5. [20] Variables relate to the Classic DA and HDA specifications, Methods to the old OPC Commands (that is now OPC UA Programs specification) and Events to the Alarms & Events (Alarms & Conditions in the new Unified Architecture). The only difference being that all this specific data types are now accessed in a unified address space.

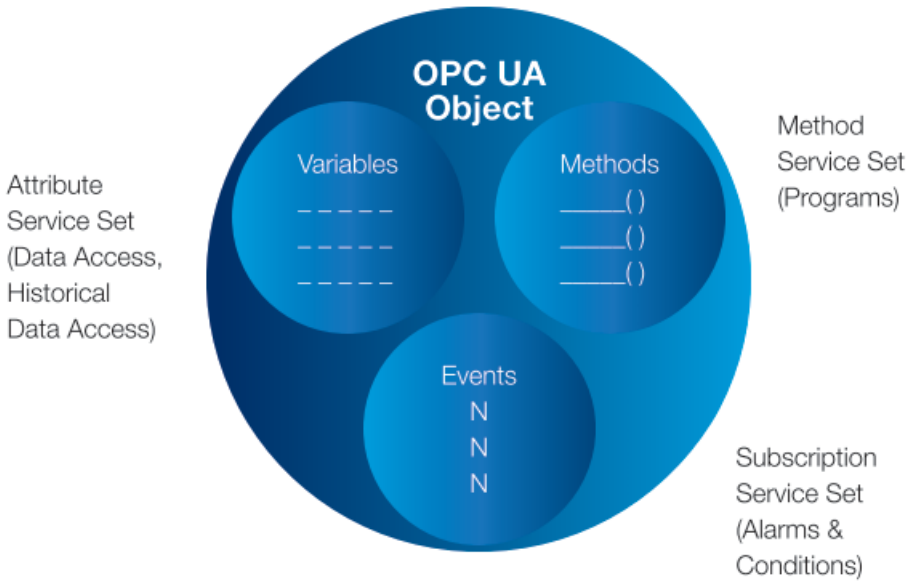

Figure 2.5: Object Model [4]

To accomplish this, OPC UA defines a meta model with Nodes and References. In OPC UA every thing is represented as a Node, and Nodes are interconnected to each other using References, to represent various kinds of relationships - spanning a network that can be organized and viewed in different ways. Nodes are composed of Attributes - a set of proprieties with data values characterizing the Nodes - and a set of it's References - Figure 2.6. Each *Attribute* has an integer id, name, description, data type and a mandatory/optional indicator. [5] Clients can access *Attribute* values using OPC UA services, mainly Read, Write, Query and Subscription/MonitoredItem Services. [5] OPC UA defines the a basic set of Attributes for each Node as it's Base NodeClass. OPC UA defines new NodeClasses by extending the BaseNodeClass. Besides the Base NodeClass, OPC UA defines the NodeClasses Object and ObjectType, Variable, VariableType, ReferenceType, Method, View and DataType - Figure 2.7. These *NodeClasses* are used to instantiate different kinds of *Nodes* and users are not allowed to define new *NodeClasses*. The *NodeClasses* have a defined set of *Attributes* and *References*. The *References* are instances of ReferenceType NodeClass. [5] Each different relationship to be defined will have its own ReferenceType. The *Node* containing the *Reference* is the *SourceNode* and the one that is referenced is called the *TargetNode*. The *ReferenceType* name represents the ReferenceType NodeClass of the *Reference*. Clients can access *References* using OPC UA View/Browsing and Querying *Services*. Variable Nodes are used to represent values. There are two Classes of Variables: Properties and DataVariables. *Properties* are characteristics of Nodes and a Property can't have other Properties associated with it to prevent recursion. All the Properties that chareacterize a Node must be defined in the same Server as the Node. *DataVariables* are associated with the contents of an *Object*. If the *DataVariable* is

complex it can have other *DataVariables* associated with it by using "HasComponent" References.

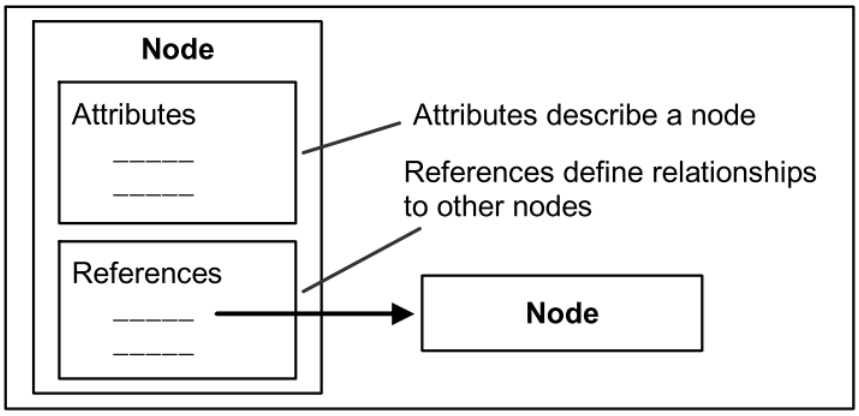

Figure 2.6: OPC UA Node Model [5]

TypeDefinitionNodes work as User defined Classes in Object-Oriented Programming. This type definition Nodes are used to provide Object and Variable Type Nodes. Object and Variable Nodes can than be instantiated accordingly to their type definition by providing a "HasTypeDefinition" Reference pointing to it's TypeDefinitionNode. OPC UA Part 5: Information Model defines the *BaseObjectType*, the *PropertyType* and the *BaseDataVariableType* as the super-type definition Node that user's can extend to create their own objects and variables typeDefinitionNodes. Complex TypeDefinitions are also possible, a TypeDefinitionNode can Reference another Node that has it's own TypeDefinitionNode without needing to Reference it's TypeDefinitionNode. This way is possible to override the default value of the Nodes Referenced in the TypeDefinitionNode. *Events* are also supported by OPC UA. The occurence of *Events* is reported by *Event Notifications* and is not directly visible in the *AddressSpace*. *Object* and *View* Nodes can be used to subscribe to *Events* by specifying the *EventNotifier Attribute*. The *Subscription* and *Monitoring* services are used to subscribe the Nodes *EventNotifications*. *Events* have associated *EventTypes*. The *BaseEventType* is the super-type for all other *EventTypes*. *EvenTypes* have no associated NodeClass and are represented in the Server *AddressSpace* as *ObjectTypes*. *Methods* are "lightwheight" functions associated with *Object* Nodes, working as methods in Class definitions in OOP or associated with *ObjectType* Nodes just as static methods of a Class. *Roles* are used to separated authentication from authorization allowing centralized services to manage the user identities and credentials and letting the Server to only worry about managing the Nodes Permissions assigned to the Roles.

"Information models follow a layered approach. Each high-order type is based on certain basic rules. In this way clients that only know and implement the basic rules can nevertheless process complex information models. Although they don't understand the deeper relationships, they can navigate through the address space and read or write data variables." [4] Some generic Information Models are: Data Access (DA), that describes the modeling of real-time data, Alarms and Conditions (AC) defines how states are handled. Historical Access (HA) defines the access to

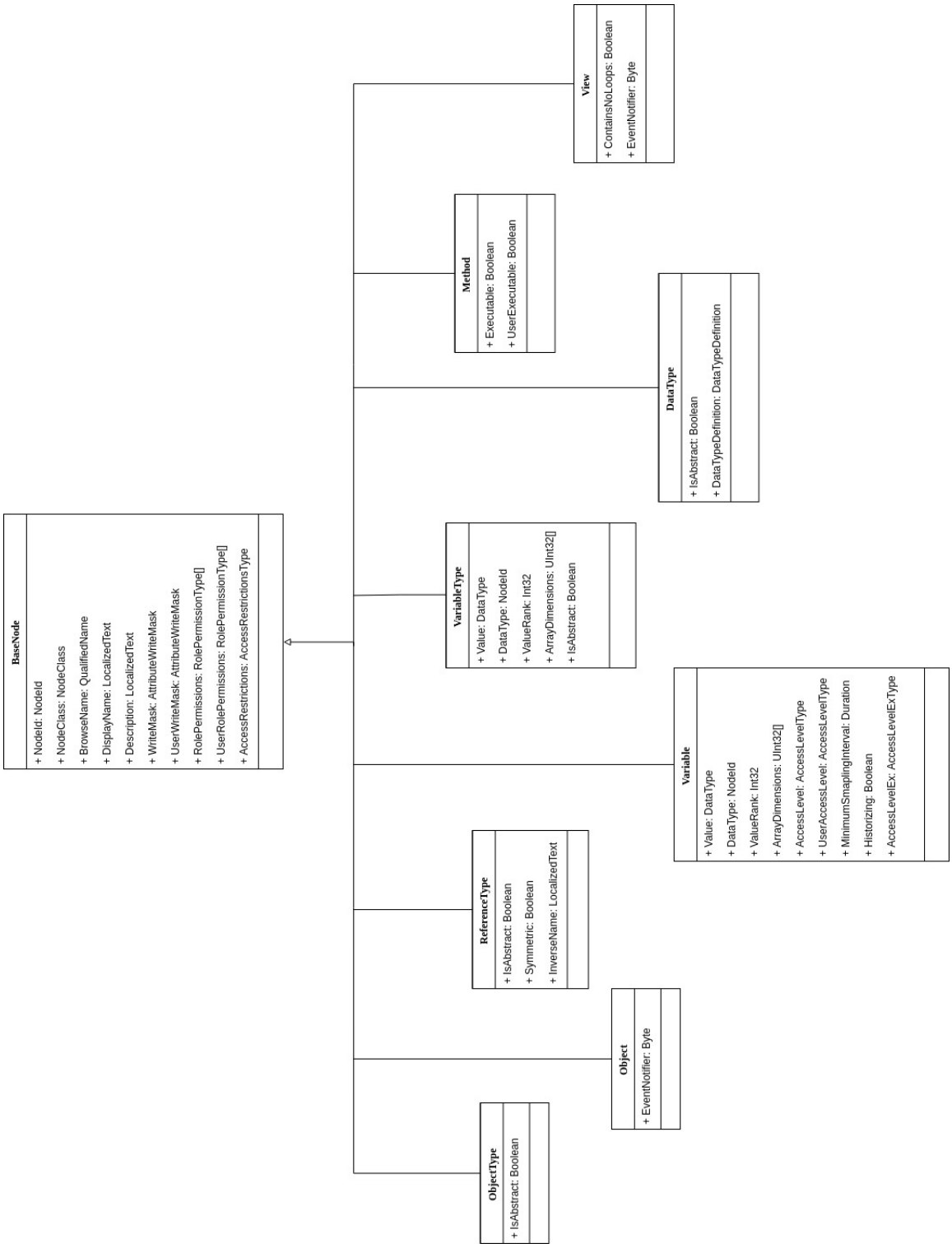

Figure 2.7: OPC UA NodeClasses

historic variables and events. Programs are represented by state machines. State transitions trigger messages to the client.

OPC Foundation collaborates with other organizations, such as standardization committees, to achieve the goal of OPC UA as a common basis for transporting data, by mapping their information models to OPC UA. This mapping rules that extend the UA information model are specified in companion standards. [30] [4]

2.2.4 Manufacturing in the Fourth Industrial Revolution - *Industry 4.0*

In Germany a platform comprised of leading software and automation companies, trade unions and politicians - Plattform Industrie 4.0 - was founded in 2012 with objective of creating a consistent framework that would fully embrace the powerful winds of the increasing *digitasation* of the economy and society, and lead German's Manufacturing Industry into the Fourth Industrial Revolution. The Germans call it the *Industrie 4.0*, where sensors, actuators and controllers link physical processes to the digital world, in what they call a *Cyber-Physical System*. This *Digital Chimeras* - half real and half digital - are connected vertically with information systems and horizontally to each other enabling the "Smart Factory". [31] [32] [33] [34] [35] The German Model inspired other *economic powers* to create similar organizations to lead the digital transformation of their Industries. By 2014 another important forum was created in the US, joining some of Information Technology's and Industry major players, namely - AT&T, Cisco, General Electric and Intel - on the similar topic of the Industrial Internet. This originated the American's platform - Industrial Internet Consortium (IIC) - aiming to promote the development, adoption and interoperability between devices, analytic engines and people in the context of the Industrial Internet of Things (IIoT). In South Korea, in June of the same year, the Manufacturing Industry Innovation 3.0 (MII3.0) was launched to bring South Korea's factories into the "Smart" Age. And in the next year, Japan published it's own vision for the Industry 4.0 - *New Robot Strategy* - and launched the *Robot Revolution Initiative* (RRI) in May, as the coordination platform for implementing their Robot Revolution. [36] In that same month, China showed the world that they too had a vision for the future and launched it's brand new initiative - *Made in China 2025* - to become the biggest automated Industry - changing production focus from quantity into quality - and challenge US's status-quo as the *de facto* economic power. [37] France and Italy, derived on the German's vision, creating the *Allieance Industrie du Futur* [38] and *Piano Industria 4.0* [39], respectively, and are working together with Platfform Industrie 4.0 in a Trilateral Commission [40] to provide interoperability in an ever more integrated digital European market. Similar initiatives were implemented in other EU countries, including Portugal with its Industria 4.0 initiative promoted by the Ministry of Economy, and the cooperation between these different initiatives is being coordinated by the *European Platform on National Initiatives on Digitising Industry*. [41] [42] [43] [44]

This various platforms are putting their goals into paper and defining reference models on how to decompose the challenges of the *digitalisation* of Industry into manageable parts. From this models, blue-prints on how to structure the Industry of the Future are being detailed. In Germany, Plattform Industrie 4.0 designed it's *Reference Architecture Model for Industrie 4.0* (RAMI 4.0)

[45] [46] and in the US's IIC created the *Industrial Internet Reference Architecture* (IIRA) [47] as the blue-print for it's Industrial Internet of Things (IIoT). Since 2015, Plattform Industrie 4.0 and the Industrial Internet Consortium, began cooperating in an attempt to aligned their different models, which was concluded in 2018 in a Joint Whitepaper - *Architecture Alignment and Interoperability*. [40] [48] The Germans and the Chinese joined forces back in 2015 and, based on RAMI 4.0, China got it's own blue-print for *Made in China 2025* - Intelligent Manufacturing System Architecture (IMSA); through cooperation both Architectures were aligned in 2016. [49] Plattform Industrie 4.0 is also working with Japan, Germany, Italy, France [50] and others to align architectures and provide the much needed standardization that will enable interoperability between the different "Smart" Industries. [51]

2.2.5 OPC UA & the *Industry 4.0*

OPC UA is being perceived by the Manufacturing Industry as the *de facto* communication protocol for building the ultimate bridge between the Enterprise and the Smart Factory - between Information Techonology (IT) and Operations Techonology (OP) - in the future Industry. [4] [52] [53] The Plattform Industrie 4.0 chose the OPC UA standardized parts - IEC 62541 - as the protocol to implement the communication layer in their reference architecture - RAMI 4.0. [54] [55] [56] In the context of the *Made in China 2025* vision, the Chinese are adopting the OPC UA standard and porting it into a National Standard [4]. South Korea is also looking into using OPC UA in their MII3.0 initiative [57]. The IIC platform is studying the use of OPC UA over Time Sensitive Networking (TSN) for the extension of Publish Subscriber communication.

Security is one of the most important factors in the "Smart Factory" of the future. OPC UA provides security mechanisms and was approved by the German Federal Office for Information Security. [4] OPC UA is cooperating with associations managing some of the most used Fieldbus Protocols in the Automation Industry, such as EtherCAT, Profinet, PowerLink, Sercos, IO-Link, [58] OPC UA is being adopted by Oil & Gas companies, to connect their systems in off-shore drills into the cloud information systems. Is being adopted by the Pharmaceutical Industry based on the OPEN-SCS initiative for connectivity and interoperability between all systems of the supply chain. [4]

2.2.6 Compatibility/Migration Path to OPC UA

The new redesign of OPC is not inherently backward compatible with OPC classic. [6]. Because of this fact OPC UA needs to provide backward compatibility with old OPC COM/DCOM based products. [30]

To accomplish this, OPC Foundation defined a three phase migration path. In the first phase wrapping components would be develop to wrap OPC COM clients and servers. In the second phase this wrappers would be customized to provide most of OPC UA capabilities to old OPC products. In the final phase, native OPC UA components would be developed and the old OPC products replaced by this new components. [30]

An OPC UA wrapper component is used to wrap an OPC COM server. [6] This wrapper is basically an OPC UA server with an OPC COM client interface, enabling OPC UA clients to access OPC COM servers. This means that the OPC UA wrapper will map OPC UA requests to corresponding COM calls and COM return values to corresponding OPC UA responses. [30]

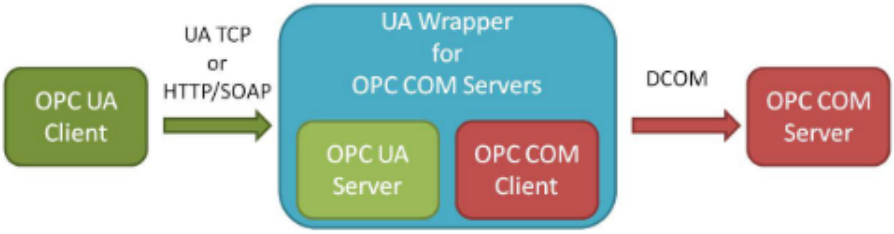

Figure 2.8: OPC UA Wrapper [6]

An OPC UA proxy component is used to wrap an OPC COM client. This proxy is an OPC UA client with an OPC COM server interface, enabling OPC COM clients to access OPC UA servers.

The OPC UA proxy will map COM calls to corresponding OPC UA requests and OPC UA responses to COM return values. [6] [30]

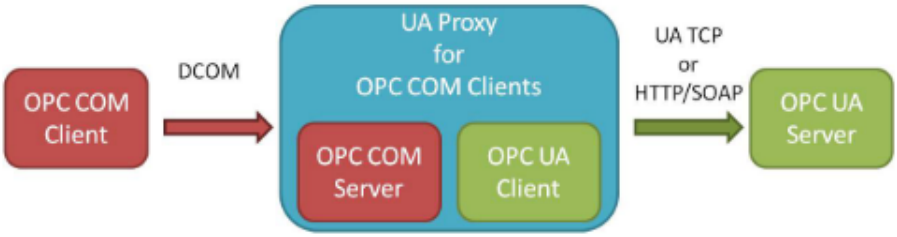

Figure 2.9: OPC UA Proxy [6]

The wrappers and proxy components are a quick and easy way to enable compatibility between OPC UA and OPC Classic products. However, some new concepts brought by OPC UA can't be mapped to OPC Classic.

The OPC UA unified address space makes variables, methods and events all available in the same address space. OPC Classic has separate address spaces for each data access specification. In this way, to map real-time data (DA), alarms and events (A&E) and historical data (HDA) from OPC classic to OPC UA a wrapper will need to be provided for each of the three OPC servers. Because of this OPC UA rich component will not be visible with wrappers and proxies. [30]

Chapter 3

State of the Art II - IEC 61131-3

In this chapter, the IEC 61131-3 specification, the third part of the IEC 61131 standard, is briefly described.

IEC 61131 is a very important standard for Industry because it defines various aspects of the Programmable Logic Controller, a digital computer specially designed for the Industrial environment. The main focus is on the third part of this standard because it defines the programming interface of the PLC and is a must know for Automation Programming Developers.

The first part of the chapter introduces the History that brought us this standard, from the relay logic circuits that were used in Industry before the introduction of the PLC to the non-interoperability problems that created the need for an Industry *grand* standard.

The second part introduces the IEC 61131 standard and is various parts and after that the IEC 61131-3 architectural model and it's programming languages will be described.

3.1 PLC History

3.1.1 Replacing Relay Systems

Before PLCs the only way to design control systems was through hard-wired circuit logic, using relays. This electromechanical control system via relays brought some challenges like huge circuits that had to be maintained, occupied lots of space, and took hours to debug and correct.

With computers evolving since the 50's the idea of using computer control systems in the industry sector began to emerge. The Hydra-Matic (Automatic Transmission) division of General Motors requested proposals for an electronic replacement of relay systems back in 1968. GM specified that this controller would have to be as versatile as a computer but at the relay systems price-range, should maintain the ladder logic concepts of programming, had to be robust to work at harsh industrial environments, and last but not least, it had to have a modular design so it could be expanded with extra modules.

3.1.2 The first PLC - *The 084*

The prize winner of this proposal was Bedford and Associates, with their PLC design, the *084*. The main designer of this PLC, Richard Morley, is considered to be "the Father of the PLC" and the legend prays that the design was detailed on New Year's Day, in 1968. After this, Bedford and Associates created it's brand new company, Modicon (Modular Digital Controller), dedicated to develop and introduce this innovative controller into the industry. [59]

3.1.3 The Firstborn - *Modicon 184*

Richard Morley's explains how the *Modicon 184*, designed by Michael Greenberg, was really the one PLC that revolutionized the market:

"The thing that made the Modicon Company and the programmable controller really take off was not the 084, but the 184. The 184 was done in design cycle by Michael Greenberg, one of the best engineers I have ever met. He, and Lee Rousseau, president and marketer, came up with a specification and a design that revolutionized the automation business. They built the 184 over the objections of yours truly. I was a purist and felt that all those bells and whistles and stuff weren't "pure", and somehow they were contaminating my "glorious design", Dead wrong again, Morley! They were specifically right on! The 184 was a walloping success, and it — not the 084, not the invention of the programmable controller — but a product designed to meet the needs of the marketplace and the customer, called the 184, took off and made Modicon and the programmable controller the company and industry it is today." [60] [59]

3.1.4 PLC Programming

During the 70's, PLC's began being adopted throughout the industry, mainly in the manufacturing industry.

The development of ladder-diagram (LD) programming - one of the prerequisites of GM Hydra-Matic proposal - was one of the factors that made the PLC accepted in the Industry. It was conceived as a replacement for hard-wired relay control systems - it followed a very similar structure of the circuit logical flow - in a way that enabled electricians with no previous training in LD to quickly understand the logic of the program.

The programs would be designed by hand using LD, translated into IL (Instruction List) - the programming language used to enter commands into the PLC memory, just like assembly language for personal computers - and then this IL program would be typed into the controller's memory.

With the introduction of microprocessors in the commercial market, and the advent of GUI (Graphical User Interfaces), in the late 70's, PLC manufacturers understood the benefits of using, this *low cost processing capability boost*, to develop graphical programming devices. [Revise]

During the coming years, PLC manufacturers, with the help of third-party software companies, would develop a multitude of proprietary systems and software, like run-time environments,

operating systems and programming languages, for their products. [61] [62] [7] [8]

This created great competition but also a *melting pot* of different languages, configuration styles and implementation methods that would difficult end-users. Programmers had steep learning curves when adopting different vendor products and no easy way to port their control programs from one vendor system to another.

3.2 IEC 61131 Standard

Because of the non-interoperability of the various ways of programming and configuring PLC's, the Industry began to try to find a way to develop standards and share knowledge between hardware and software manufacturers. This would introduce interoperability between the different vendor's products, and avoid the risk of manufacturers pouring huge amounts of money on software development and testing that wouldn't be accepted on the market. Instead, the industry would try to build their products upon common software solutions.

The IEC 61131 was the Industry *check-list* for this vision. The standard extends various previously established standards (IEC 50, IEC 559, IEC 617- 12, IEC 617-13, IEC 848, ISO/AFNOR, ISO/IEC 646, ISO 8601, ISO 7185, ISO 7498) into a grand framework that encompasses all aspects of PLC's - mechanical, electrical and logical - and establishes a set of main rules that vendor's can adopt in their effort for portability. [7] [8] [63]

3.2.1 IEC 61131 Parts

The IEC 61131 was conceived as a 5 part standard but other parts were later added. [64]

- **IEC 61131-1**: *General information*

  Part 1 of the IEC 61131 standard contains general definitions and typical functional features which distinguish a PLC from other systems.

- **IEC 61131-2**: *Equipment requirements and tests*

  Part 2 of the IEC 61131 standard defines the electrical, mechanical and functional demands on the devices as well as corresponding qualification tests.

- **IEC 61131-3**: *Programming Languages*

  Part 3 of the IEC 61131 standard defines a set of programming languages for the PLCs.

- **IEC 66131-4**: *User guidelines*

  Part 4 of the IEC 61131 standard is a TR intended to be a guide for PLC user's in all phases of an automation project.

- **IEC 61131-5**: *Messaging service specification*

Part 5 of the IEC 61131 standard defines communication specifications for PLC to communicate with each other and with other factory devices.

- **IEC 61131-6**: *Functional Safety*

  Part 6 of the IEC 61131 standard defines safety-related procedures for programmable controllers and associated peripherals.

- **IEC 61131-7**: *Fuzzy Control Programming*

  Part 7 of the IEC 61131 standard defines basic programming elements for the use of fuzzy logic control in PLCs.

- **IEC 61131-8**: *Guidelines for the application and implementation of programming languages*

  Part 8 of the IEC 61131 standard is a TR that provides developers with a guide for the programming languages defined in IEC 61131-3.

- **IEC 61131-9**: *Single-drop digital communication interface for small sensors and actuators (SDCI)*

  Part 9 of the IEC 61131 standard extends the traditional digital input and digital output interfaces defined in Part 2 towards a point-to-point communication link.

3.3 IEC 61131-3

This specification started being drafted in 1982 and was published in December 1993 as the third part of IEC 61131 standard. A second version of the standard was published in 2003 and the third and current version in 2013.

IEC 61131-3 standardizes the programming interface. It provides a set of guidelines for the programming of PLCs, defining the structure of logical programming blocks and the syntax and semantics of a set of programming languages. There's five programming languages defined: ladder-diagram (LD), instruction list (IL), structured text (ST), function block diagram (FBD) and sequential function chart (SFC). The set of different programming languages enables the development of control programs for a wide specter of programmers with different backgrounds and the re-use of software blocks across projects. The defined structure makes it possible to divide the project into manageable units, that can be handled by different programmers, bringing a faster project development flow. The strong set of data typing rules reduces error proneness.

All this benefits make IEC 61131-3 a standard that provides various advantages for end-users, software developers and manufacturing suppliers, reducing waste of time and resources and enabling each different user of the standard to focus on what they are best at. Manufacturers don't need to follow all rules of this specification. For the programming package of a vendor's product to be compliant with the standard, it needs to support at least one of the five programming languages defined. Vendors will try to implement some of this guidelines in their products, and customers

will be able to access how compliant the products are, and choose what best fits their needs. [63] [7] [65]

3.3.1 Software Model

The Software Model defines the structure of the software, describing the logical programming blocks and the relations between them.

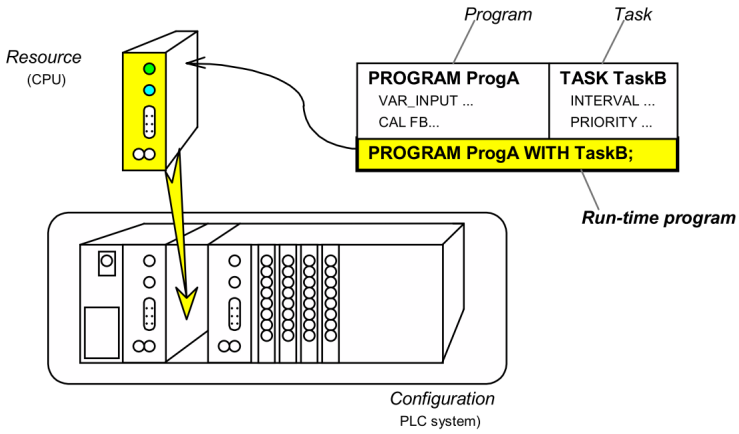

Figure 3.1: Configuration Elements [7]

The *Configuration* is the higher programming block. It represents the PLC system. One *Configuration* can have one or more *Resources*. Each *Resource* represents a central processing unit (CPU) of the PLC. A *Resource* executes one or several *Tasks*, just like a CPU executes various processes. Each *Task* will have one or more *Programs* assigned and their run-time properties defined. A *Program* can be structured using *Functions* and *Function Blocks*. [14] [7] [8]

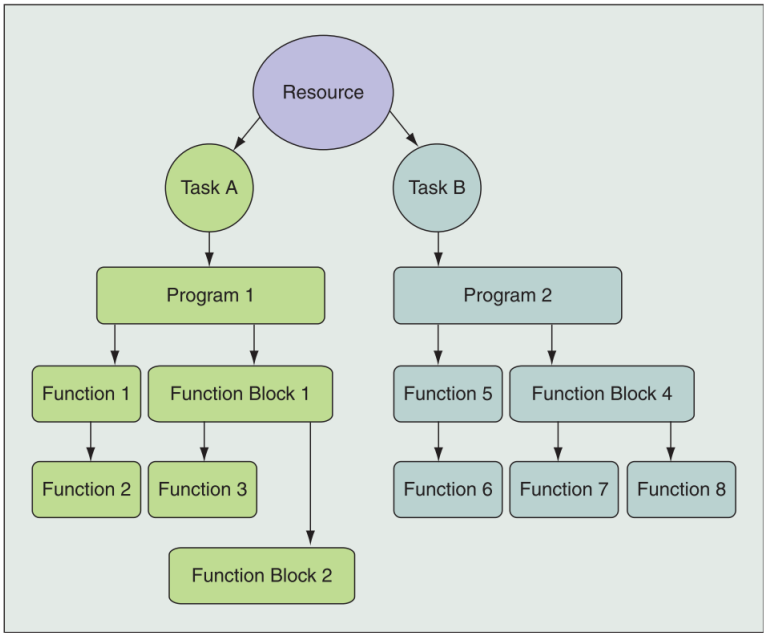

Figure 3.2: Software Model [8]

In a *Configuration* global *variables* can be declared, and are accessible for all the lower software blocks declared inside the *Configuration*. In the same way, *Resources* support the declaration of global *variables*, accessible by *Program Organization Units (POU)* assigned to *Tasks* in this *Resource*.

Configurations and *Resources* can be started and stopped via the appropriate functions defined in IEC 61131-1. When a *Configuration* starts, all its global *variables* shall be initialized, followed by the starting of all *Resources* in the *Configuration*. When the *Resource* of a *Configuration* is initialized, all its *variables* shall be initialized, followed by the starting of all its *Tasks*. [14] [7]

3.3.2 Communication Model

The Communication Model describes how the various logic elements of IEC 61131-3 communicate with each other.

Inside a *Program* variable values can be communicated directly by connection of the output of one program element to the input of another. This connection is shown explicitly in graphical languages and implicitly in textual languages.

Variable values can be shared between *Programs* in the same *Configuration* by declaring those variables as *global variables* in the *Configuration* and as *external variables* in the *Programs*.

Variable values can also be communicated between different parts of a *Program*, between *Programs* in the same or different *Configurations*, or between a programmable controller program and a non-programmable controller system, using the communication function blocks defined in IEC 61131-5.

Programmable controllers and non-programmable controllers can also transfer data through access paths, using the communication mechanisms defined in IEC 61131-5. [14] [7]

3.3.3 Program Organization Units

The Program Organization Unit (POU) is the smallest independent software unit of a user program.

The structure of a Program Organization Unit consists of three parts. The *POU declaration*, where the type and name of the POU are specified, and in the case of *Functions* also the data type. The *variables declaration*, where the variables used in this POU are declared and distinction between types of variables is specified. And finally, the *body* of the POU, where the logic circuit or algorithm is described. [7] [14]

The standard defines three types of program organizations units: *Function*, *Function block* and *Program*: [7] [14]

Function is defined as a program organization unit which returns exactly on data element, the function result, and an arbitrarily number of additional output elements. Functions have no internal state, which means that when invoked with the same input arguments a function must always return the same result value and output parameters values.

```
FUNCTION SUM: INT;
    VAR_IN_OUT A: ARRAY [*] OF INT; END_VAR;
    VAR
        i, sum2: DINT;
    END_VAR;

    sum2:= 0;
    FOR i:= LOWER_BOUND(A,1) TO UPPER_BOUND(A,1)
        sum2:= sum2 + A[i];
    END_FOR;
    SUM:= sum2;
END_FUNCTION
```

Listing 3.1: Example Function declaration in ST [14]

Function Block is defined as a program organization unit which, when executed, yields one or more values. Functions blocks have internal variables, which shall persist from one execution to another, meaning that the output values may yield different results even when called with the same input arguments. Only the input and output variables shall be accessible outside of function block instance, the internal variables shall be hidden from the user of the function block. Function blocks can be instantiated within programs or other function blocks, and can be used as the input of a function or another function block.

```
FUNCTION_BLOCK ConvergeC
  VAR
    xx : INT;
  END_VAR
  VAR_INPUT
    p : BOOL;
  END_VAR
  VAR_OUTPUT
    xp : BOOL;
    xn : BOOL;
  END_VAR
  VAR_INPUT
    recv1 : BOOL;
    recv2 : BOOL;
    frp : BOOL;
    frn : BOOL;
  END_VAR
  VAR_OUTPUT
    take : BOOL;
    free_o1 : BOOL;
    free_o2 : BOOL;
    rp : BOOL;
    rn : BOOL;
  END_VAR
  VAR_INPUT
    free_i : BOOL;
    pulldir : BOOL;
  END_VAR

  (* Body goes here *)

END_FUNCTION_BLOCK
```

Listing 3.2: Example Function Block declaration [14]

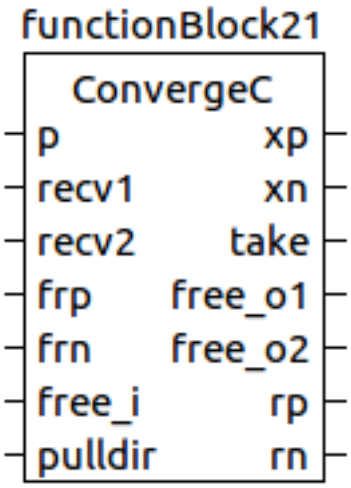

Figure 3.3: Function Block for the ConvergeC FBD representation

Program is a POU that represents the "Runtime Main Program". It is defined by IEC 61131-1 as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system". All variables of the program, that are assigned to physical addresses, must be declared inside this POU or above it. Programs can only be instantiated within resources.

```
PROGRAM program0
  VAR
    LocalVar0 : BOOL;
    LocalVar1 : R_TRIG;
    LocalVar2 : ConvergeC
  END_VAR

  (* Body goes here *)
END_PROGRAM
```

Listing 3.3: Example Program declaration in ST [14]

3.3.4 Programming Languages

IEC 61131-3 defines five programming languages, from this set, IL and ST are textual languages, and, LD, FBD and SFC, are graphical languages.

Because SFC can also be expressed in a textual format, we can also say that IEC 61131-3 has 6 languages: 3 textual and 3 graphical languages. Being that SFC is counted in the textual

languages with it's text version and in the graphical languages with it's graphical version [7].

As defined in [14] and explained in [62]:

Instruction List (IL) is a low-level language, similar to assembly, with a simple instruction set. It was used to write the commands into the PLC memory and can be easily ported between hardware platforms. Because of it's text nature and compactness, this language runs faster than graphical languages and occupies less memory space - an advantage not that useful with the huge speed of CPUs and large memory in modern PLCs. On the downside, because of it's low-level nature, this language is not suitable for structured programming and complex functions.

```
A:  LD      %IX1    (* PUSH BUTTON *)
    ANDN    %MX5    (* NOT INHIBITED *)
    ST      %QX2    (* FAN ON *)
```

Listing 3.4: Example IL code

Structured Text (ST) is a high-level language, similar to PASCAL and C, that embraces the growing complexity of PLC programming. It's a very flexible language and like major computer programming languages, it's fairly easy to structure your code. For most newcomers into the industry, that have better computer programming skills than electrical wiring ones, this language is the go-to for PLC programming.

Ladder-diagram (LD), invented in the U.S., evolved from a schematic method of designing relay racks into a programming language. It resembles a series of control circuits and was fairly simple to use because it was easy for non-trained personnel with electrical background to understand the logic of the program. It's one of the most used IEC 61131-3 languages and it's ideal for simple control applications. In large programs the ladder grows extensively and can be very difficult to interpret and troubleshoot, and it's not very suitable for implementing full processes either.

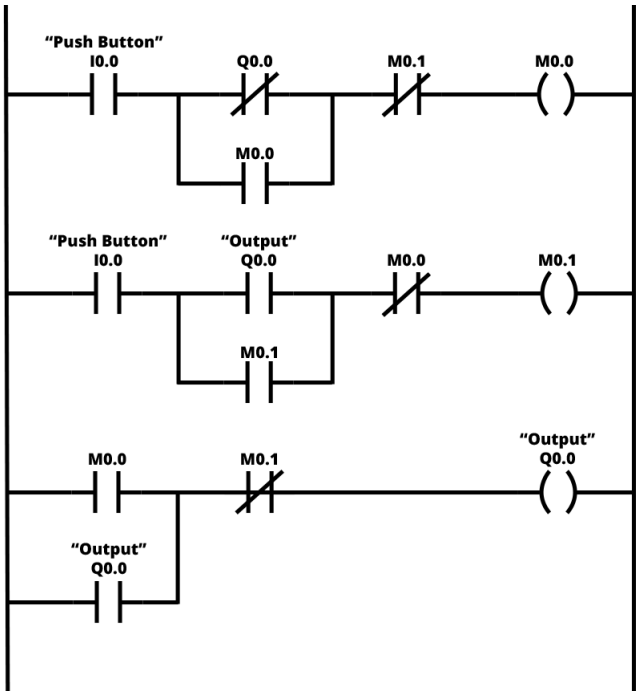

Figure 3.4: Example of a LD program

Function Block Diagram (FBD) is a graphical language based on the interconnection of Function Blocks. Like LD it also resembles a wiring diagram because the blocks are "wired" into a sequence chain and it's fairly easy to follow the logic flow of the program. Besides that, with the ability to use precoded blocks, it's easy to incorporate encapsulated logic into FBD.

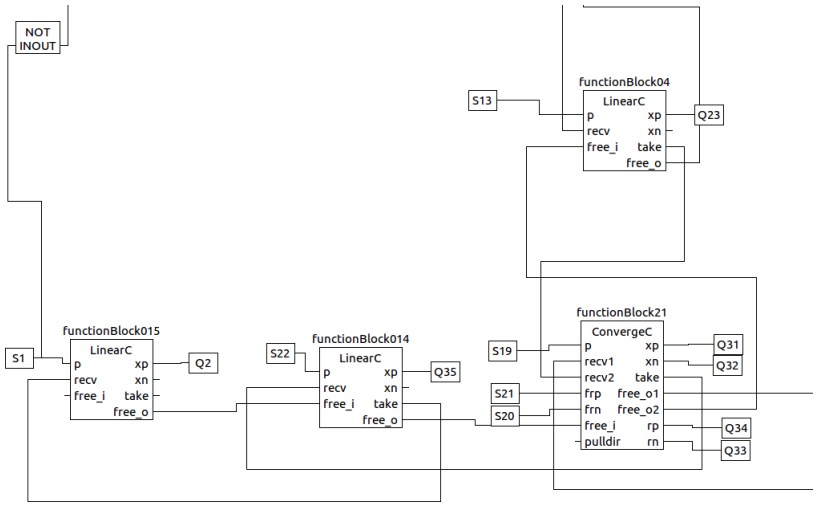

Figure 3.5: Snippet from a FBD program

Sequential Function Chart (SFC) is a language based on the Grafcet language - that was itself based on Petri nets - and is the main IEC 61131-3 language for sequential or state-based control. Beside it's graphical version, SFC can also be expressed using a textual version.

In SFC, control tasks are broken into steps, and steps are connected to each other with transitions. Each step executes an action and stays active until the transition to the step below activates. Due to it's graphical nature, it's simple to understand and follow the sequential behavior of the program. This programming language is ideal for applications with repeatable processes, however, because of the complexity of the structuring of this programming language, without a previous well defined design, it can become difficult to troubleshoot. Besides that, it is not possible to write a complete program using only SFC, which makes it a kind of a pseudo-programming language.

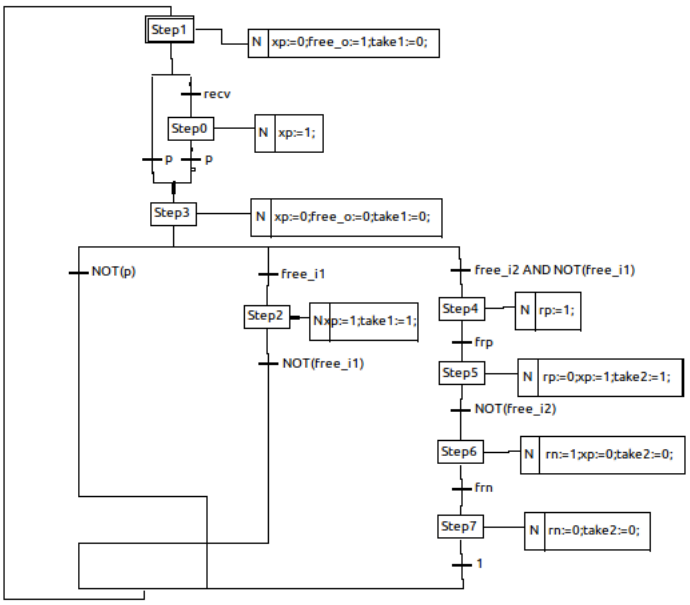

Figure 3.6: Example of a SFC FB declaration

3.3.5 IEC 61131-3:2013

The third version of IEC 61131-3, published in 2013, brought advanced Object-Oriented Programming capabilities to the standard.

Addition of methods, that behave like traditional functions, to the Function Blocks helps structure the function block by encapsulating the behavioral logic into methods and clearly specify the data associated with them.

It introduces the concept of inheritance by providing declaration of Classes, that can be extended. Classes can also be declared as abstract to provide base type object definitions, can implement methods and can be instantiated resulting in objects.

The addition of Interfaces provides a way for grouping abstract method declarations. Interfaces also have inheritance and multiple Interfaces can be extend simultaneously. This concept is used to specify the methods a class should implement by inheriting the Interface.

With the introduction of pointer references, the IEC 61131-3 brings the concept of Polymorphism, providing a way to implement methods that work with not just one base type but also subtypes of it.

Another main addition was the concept of namespaces, that provides a way for organizing variables and sub-routines and is used in various Object-Oriented Programming languages. This concept enables IEC 61131-3 programmers to create collections of POUs or data types avoiding identifier ambiguities.

With introduction of this Object-Oriented concepts, IEC 61131-3 programmers have now more options in terms of automation programming. By providing them with the same tools most programming languages have at their disposal nowadays it enables re-usability and improves maintenance of software.

Chapter 4

State of the Art III - Beremiz

In the first part of this chapter, Beremiz, an IEC 61131-3 compliant open-source IDE, is described. A brief overview of the technologies that enabled the development of this project is also made. After that, the compiliation process and the most important Plugins are described.

The second part follows on the Beremiz Compilation Process and describes in detail the MatPLC IEC 61131-3 Compiler (MatIEC), the back-end of the Beremiz IDE. First the Architecture of Compilers is specified and after that, those concepts are used to describe how MatIEC Architecture is modeled and how it relates to the Compilers Architecture.

4.1 Beremiz IDE

The mastering of the programming practices and the use of the Programmable Logic Controller, requires a lot of time and testing. This systems are expensive and errors in programming can cause system failures and a costly down-time. To recreate Industrial scenery, simulation of PLCs and Industrial environments can be run in low-cost PC architectures to train new personnel without the risks and costs of using real systems. Various of this simulation and development environments were created by many of the PLC manufacturers but they were still proprietary with expensive licensing costs.

Beremiz changes that, it is a free and open-source integrated development environment for the IEC 61131-3 defined programming languages. Beremiz provides students with an IEC 61131-3 framework were they could play and learn at their own pace, practicing and developing their programming skills.

This framework consists of a GUI (PLCopen Editor) and a backend-compiler (MatPLC's IEC compiler). The GUI has editors for all the five languages and let's user's create IEC 61131-3 compliant programs, and imports and exports the programs accordingly to PLCopen standard TC6-XML Schemes, allowing the projects to be exchanged between PLCopen standard compliant IEC 61131-3 editors. The back-end compiler converts the program written by the user into an equivalent C program that can be executed in various processor architectures.

Beremiz is a community effort and is constantly being updated with new functionality, provided by the creation of Plugins. [66] [63]

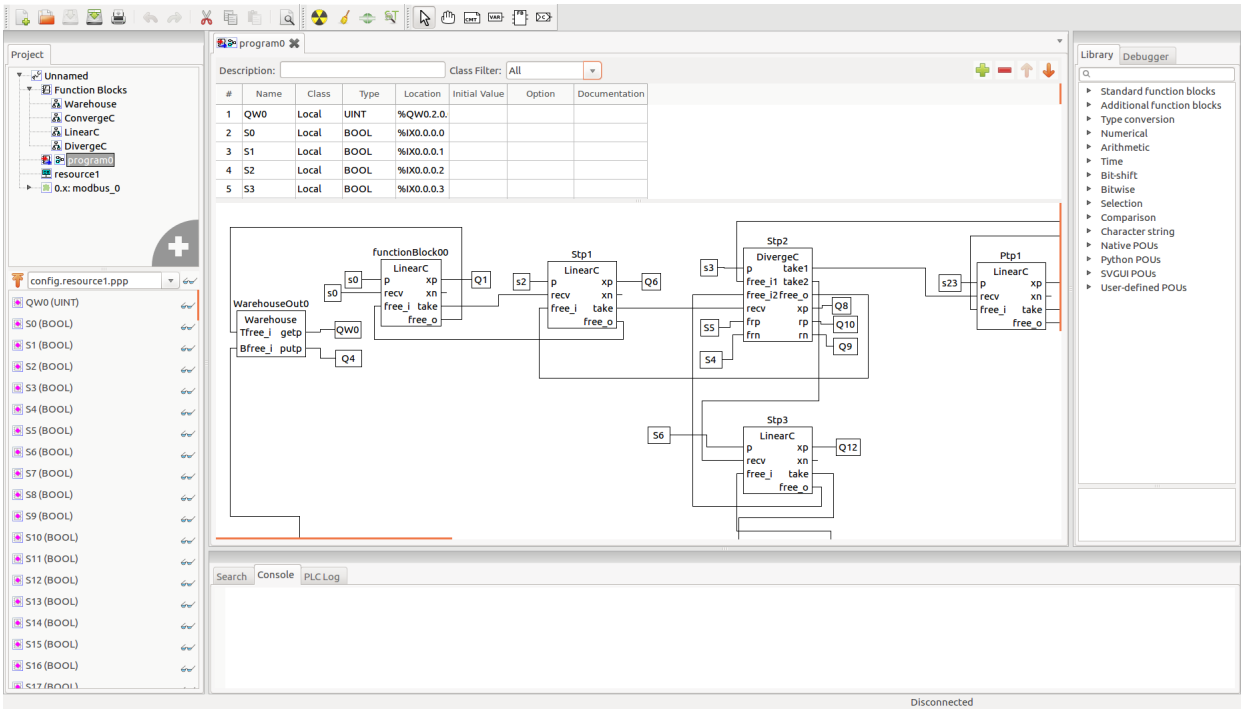

Figure 4.1: Beremiz IDE

4.1.1 PLCopen & XML

IEC 61131-3 brought a new level of interoperability to the Control Industry, enabling users to use standardize programming languages and decouple themselves from proprietary software options. However, the exchange of libraries and projects between IEC 61131-3 compliant IDEs was not established in the standard. PLCopen, an organization devoted to IEC 61131-3 standard, resolved this issue creating a specification that enabled the exchange of IEC 61131-3 information using XML technology.

This specification was published in 2005 and provides the ability to transfer IEC 61131-3 complete or incomplete projects from one IDE to another, without loss of information. It provides a set of XML Schemes for mapping the IEC 61131-3 elements. When converting from IEC 61131-3 graphical languages to XML, information like the place and position of the blocks, and it's connections, are also stored. It also stores comments, user derived datatypes and POUs, and the structure of the project.

Besides the main goal of importing and exporting projects from one IDE to another, it also provides an exchange format between all five languages, and it serves as an interface for IEC 61131-3 support tools, like simulation, modeling or documentation tools. [67]

4.1.1.1 GUI - PLCopen Editor

The **PLCopen Editor** was created mainly by Edouard Tisserant and Laurent Bessard for the Beremiz project. The OpenPLC is capable of running Structured Text (ST) programs. The PLCopen Editor is written in python, using wxPython, the python binding to wxWidgtes, - a C++ library that enables developers to create applications that work in all major platforms - making it a cross-platform application.

This graphical interface let's users create a project composed of several POUs that are listed in a tree view panel. This tree view let's users expand the POU to expose their children, like interface and internal variables. This tree view also enables users to add other tools to their projects, provided by the plugins, like the Canfestival CanOpen support or the Modbus support.

The PLCopen Editor has editors for all the five IEC 61131-3 languages, and they all follow the MVC (Model-View-Controller) [68] paradigm. The Model, composed of it's various object classes, is dynamically generated from the PLCopen XML Scheme. This enables future changes to the PLCopen specification to be added automatically.

The graphical editors were designed in a way to prohibit the user from introducing illegal layout. This way, even if the program is incomplete it will always be in a correct state. The textual editors include syntax highlighting, auto completion and syntax error highlight. [63]

4.1.2 Compilation Process

After the user writes is program using the User Interface, he can simulate the program running, using the Beremiz softPLC, by clicking on the button that initiates the compilation process. During the compilation process, the graphical programs, FBD or LD, are converted to ST; SFC is described in it's textual format; and ST and IL code doesn't need to be converted. Beremiz builds the complete project into a text format file. This file, representing the PLC program, is translated by the back-end compiler into C code. The C code is further compiled by the gcc compiler, translating the PLC program C code into an executable software PLC. The softPLC is then executed and the simulation starts running.

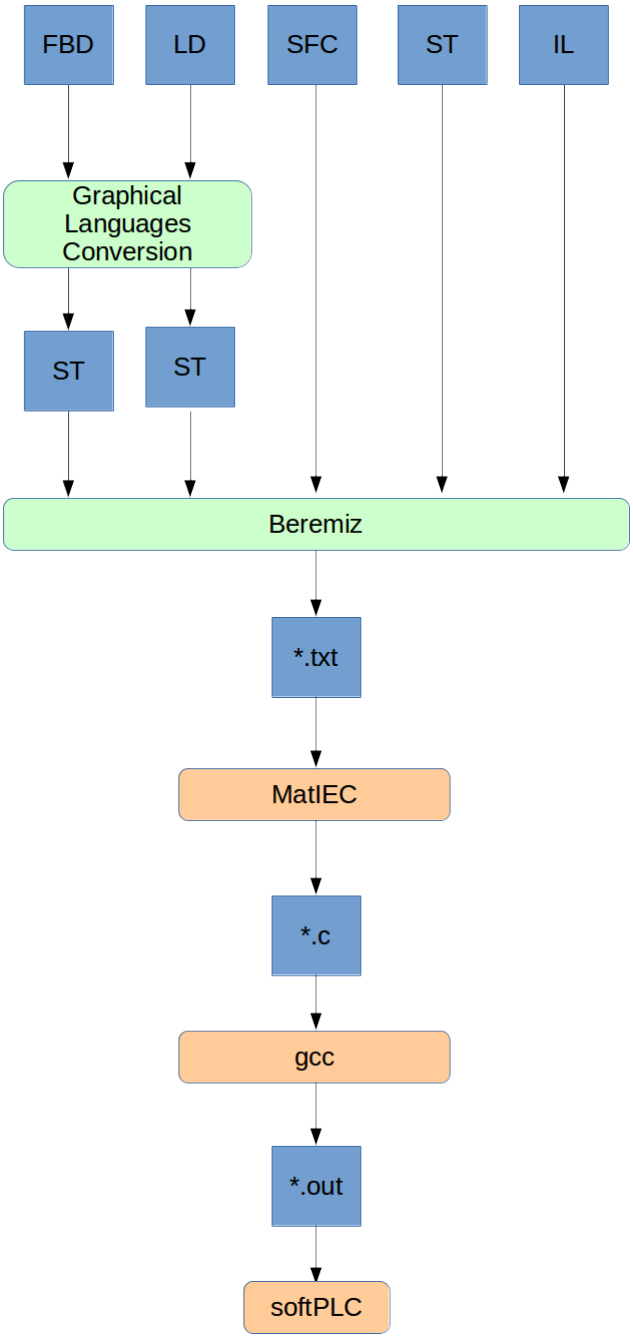

Figure 4.2: Beremiz Compilation Process

4.1.2.1 Conversion of Graphical Languages

A module is responsible for translating the IEC 61131-3 graphical languages (FBD and LD) into ST using the reverse propagation algorithm [69].

The IEC 61131-3 graphical state machine language (SFC) is converted to it's textual version.

After all the graphical languages have been converted to ST, and SFC to it's textual format, this languages will be compiled into an equivalent C program using the backend-compiler. [63]

4.1.2.2 Back-end Compiler

The back-end compiler is implemented using MatPLC's IEC compiler. This compiler, and compilers architecture, will be described in the next section - 4.2 MatIEC Compiler.

4.1.3 Plugins

4.1.3.1 CanFestival

CanFestival - a free software CANopen framework - created a plugin that provides Beremiz with a CanOpen interface to physical I/O. [70]. *CANopen* is a high-level communication protocol and device profile specification - based on the fieldbus CAN (Controller Area Network) protocol - that abstracts the user from hardware-specific details. It implements the top five layers from the OSI model. CAN is a reliable multi-master serial bus system, originally developed for in-vehicle network, that as spread to other industries. [71] [72]

4.1.3.2 svGUI

The SciViews **svGUI** package provides functions to implement GUI (Graphical User Interface) in R. It is independent from any particular GUI toolkit, centralize info about GUI elements currently used, and dispatch GUI functions to the particular toolkits used [73].

This plugin enable user's to draw their HMI using standard SVG drawing tools. Using wxSVG it's possible to render the SVG elements using wxWidgets library. The link between the SVG elements and the wxWidgets objects is made using a XML file.

4.1.3.3 Modbus

Beremiz now has support for the Modbus protocol. Modbus is one of the oldest fieldbus communication protocols, created by Modicon back in 1979, based on Master-Slave topology, and was previously just used on wired serial communications but extensions were created to provide physical interface for TCP/IP networks and wireless communications. Modbus is considered to be a communication protocol "ideal for quick, reliable communications of simple data to and from I/O devices, while consuming low bandwidth" [74], and is still widely used in Industry today.

4.1.3.4 BACnet

Mário de Sousa recently created a Plugin to provide Beremiz with BACnet support. BACnet - *Building Automation Control Networks* - is a widely used communication protocol in BAC (Building Automation and Control) systems - the combination of hardware and software systems that control all the different systems in buildings from power consumption, ventilation, elevators to security and others - essential for the development of intelligent and energy efficient buildings and their connection to the Smart-Grid. [75]

4.2 MatIEC Compiler

4.2.1 MatPLC

Mário Jorge Rodrigues de Sousa, started the Machine Automation Tools Programmable Logic Controller (MatPLC) back in 2002. Since user-friendly both, fully-fledged, PC based architectures - Industrial PC's - and hybrid PC/PLC PAC's (Programmable Automation Controller) were used alongside the trustworthy PLCs in the factory-floor - the PLC manufacturers *vendor-lockin* could be challenged - by creating an open-source software PLC, that could be run in PC architectures, providing students all-over the world with a free emulated PLC.

The software architecture was designed in a modular way, and can be considered to be both a PLC and a SCADA package, extending the traditional functionality of both. This modular architecture contrasts with the infinite loop in which traditional PLCs run, providing a way for this autonomous modules to be developed simultaneously by different teams. [76]

MatPLC would die and be reborn from the ashes like a Phoenix by the Beremiz project - where the MatPLC IEC Compiler is used - continuing to fly the open-source flag high and proud.

4.2.2 Compilers

Before a software program can be run in a computer, it has to be first translated into a computer-compatible form to be executed.

Compilers are software systems that read a program in a *source* language and translate it to an equivalent program in the *target* language. It also provides feedback of any errors encountered in the source program during the translation process. If the program translates a high-level language into a machine-language it's usually called a *Compiler*, if it translates one hihg-level language into another one, it's called a *Code Translator*, and if it translates and executes each portion of the code in a sequence, it's usually called an *Interpreter*. [77]

4.2.2.1 Phase & Pass

A compiler can have many phases and passes [9]:

- **Pass**: A pass refers to the transversal of a compiler through the entire program.
- **Phase**: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase. It can also be referred to as a stage of the compiler.

4.2.2.2 High-level Structure

The language mapping done by compilers can be divided in two main parts [77]:

The **Analysis** part of the compiler, known as the front-end of the compiler, is responsible for reading the source program, dividing it into smaller parts and matching these parts against the grammatical structure imposed by the compiler, checking for lexical, grammar and syntax errors. If errors are detected in the source program, informative messages must be provided to help the user correct their program. It also collects information about all the *symbols* found in the source program and maintains it in a data structure called *symbol table*. If no error is detected an *intermediate representation* of the source code, another data structure, is formed from the source code grammatical structure and fed to the *Synthesis* part alongside the *symbol table*.

The **Synthesis** part, the compiler back-end, constructs the target program with the help of *intermediate source code representation* and the *symbol table* created in the *Analysis* part.

4.2.2.3 Low-level Structure

At a more detailed level we can divide the compilation process in a sequence of phases instead of just dividing it in two main parts. In a typical compiler it's main phases will be [77] [10] [9]:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Machine-Independent Code Optimization
6. Code Generation
7. Machine-Dependent Code Optimization

Since optimization is optional, one or the two optimization phases may not be implemented by the compiler [77] and each compiler will have a different set of phases, depending on it's inner workings.

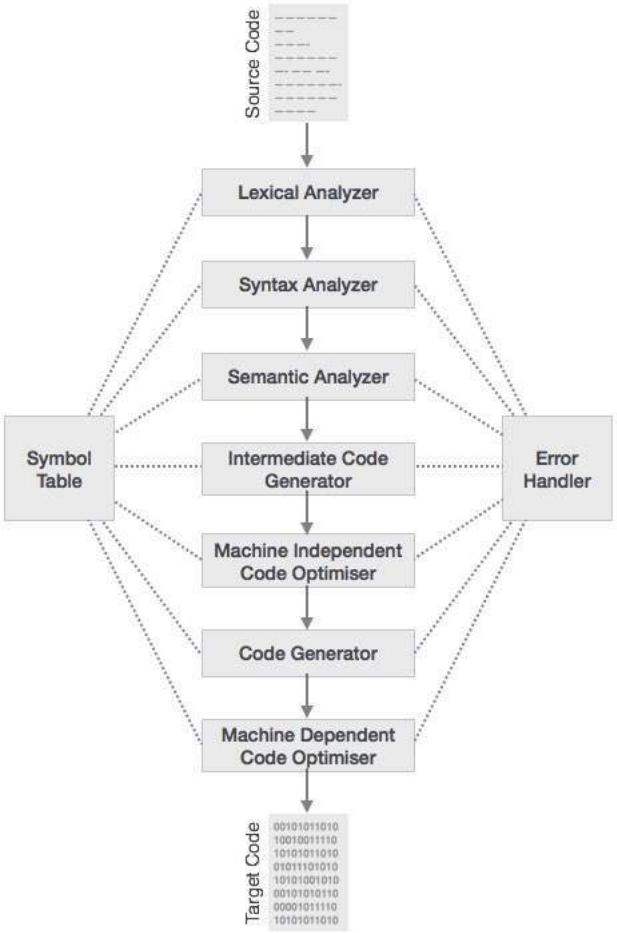

Figure 4.3: Compiler Phases [9]

4.2.2.4 Lexical Analysis

The *lexical analyzer* or *scanner* reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes*. This *lexemes* are represented in the form of *tokens*:

<token-name, attribute-value>

The token-name is an abstract symbol and the attribute-value points to an entry in the *symbol table* for this token. This tokens are then passed to the *Syntax Analysis* phase.

4.2.2.5 Syntax Analysis

The *syntax analysis* or *parsing* phase converts the tokens defined in the previous phase and generates a tree-like intermediate representation, an *abstract syntax tree*, that depicts the grammatical structure of the token stream.

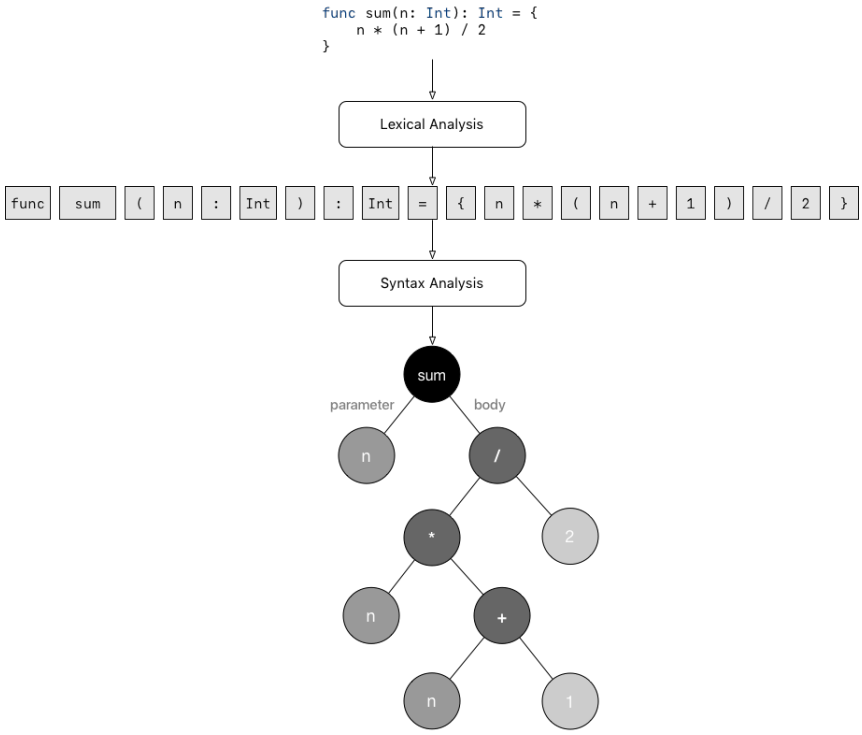

Figure 4.4: *Abstract Syntax Tree* [10]

4.2.2.6 Semantic Analysis

The *semantic analyzer* uses the syntax tree and information in the symbol table to check if the source program is consistent with it's semantic rules, checking assignment of values, keeping track of identifiers, their types, and expressions. This phase annotates this information in the *syntax tree* and passes it to the next phase.

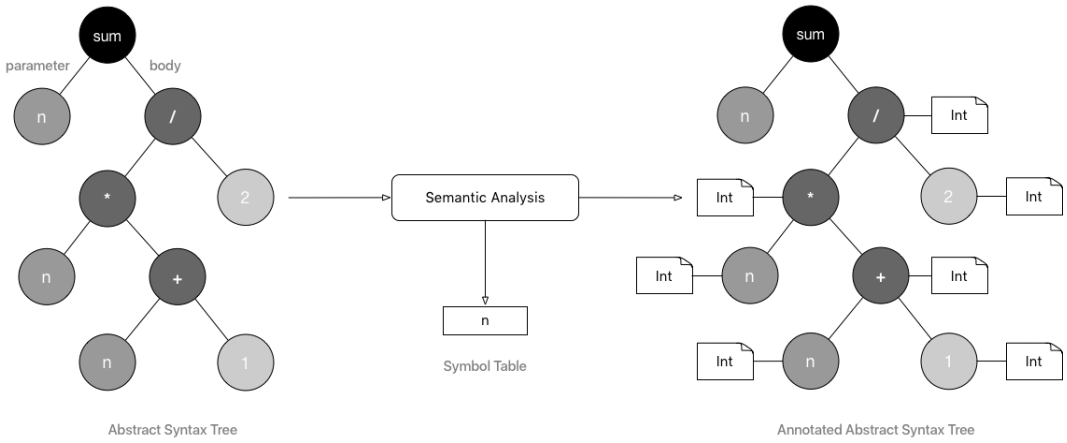

Figure 4.5: *Annotated Abstract Syntax Tree* [10]

4.2.2.7 Intermediate Code Generation

In this phase the compiler generates a machine-like intermediate representation, something between the source-language and the machine-language. This intermediate representation should be easy to translate to a specific target machine code. This is the last phase of the front-end of the compiler.

4.2.2.8 Code Optimization

The optimization phase improves the intermediate code generated earlier to improve performance. It can make the code faster, shorter or even try to reduce the power consumption. In a Machine-Independent Optimization phase the compiler improves this code in a general way, and in the Machine-Dependent Code Optimization it tries to improve the code generated in the *Code Generation* phase for a specific machine, taking it's specifications into account.

4.2.2.9 Code Generation

This phase takes the *intermediate representation* of the source code and maps it to the target language.

4.2.3 MatIEC

Machine Automation Tools for IEC 61131-3 (MatIEC) is a code translator for the programming languages defined in the IEC 61131-3 standard.

This compiler is compliant with the second version of the standard and supports all three IEC 61131-3 defined textual languages: IL, ST and SFC textual version. It translates this languages to ANSI C code or to the same source language for debug purposes of the front-end phases of the compiler. All POU parameters and variables are accessible through nested C structures and located variables are declared as extern C variables.

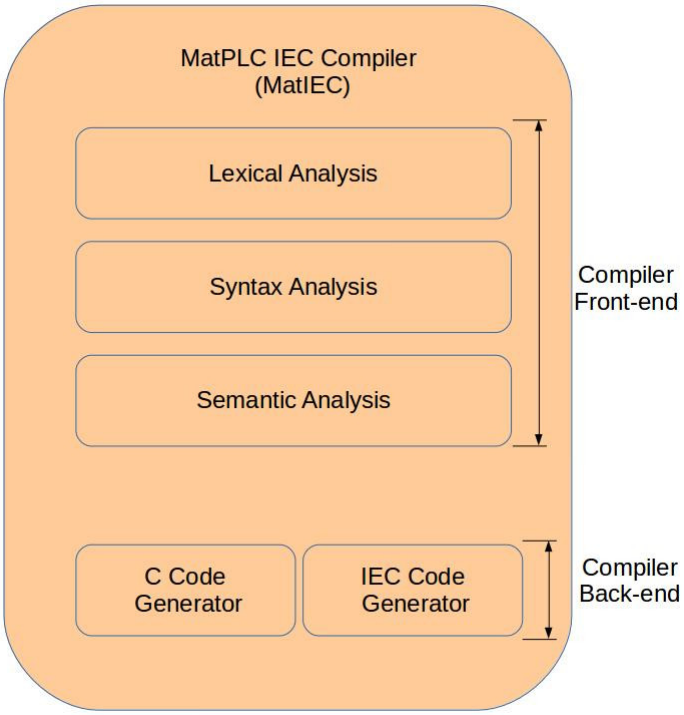

Figure 4.6: MatIEC Compiler Structure

This translator implements 4 phases of the compilers architecture: Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Generation - where the ANSI C code is generated. To become a full fledged compiler, translating from the IEC 61131-3 textual languages to machine-code language, one last phase of code generation - Machine Code Generation - is implemented using an external compiler like gcc.

This architecture enables the addition of more types of code generation for any output language without the need to rewrite any of the previous stages. [63]

This compiler uses Flex (Fast Lexical Analyzer Generator) to implement it's Lexical Analysis Stage. Flex is an open-source and faster alternative version of lex, a lexical analyser generator written by Mike Lesk and Eric Schmidt and described in 1975. [78]

For it's second phase, Syntax Analysis, MatIEC uses Bison, a general-purpose parser generator, distributed under the GNU license. Bison is the open source descendant of yacc, a parser generator written at Bell Labs and popular among users of Unix systems. [78] [79]

This first two phases are executed in a single pass, this is done because the second phase feeds data back to the first phase.

In the third phase, Semantic Analysis, there is a pre-phase, executed in a single pass, that populates the symbol tables. After that, another pass will execute the flow control analysis and another one will execute the data type analysis. The flow control and the data type analysis will annotate the abstract syntax tree with this information. The abstract syntax tree has been implemented as a tree of objects that follows the visitor pattern, enabling the addition or removal of stages without needing to re-edit the abstract syntax tree.

The last phase is the Code Generation phase and is responsible for generating the target language and it is done in a single pass.

Possible additions to the architecture include the code optimization phase. [63]

Chapter 5

OPC UA Information Model for IEC 61131-3

In this chapter, the Specification *OPC UA Information Model for IEC 61131-3*, used in this project as the model for the mapping between IEC 61131-3 and OPC UA, is described alongside it's parent specifications and it's mapping to XML. After that, a brief example on how to use this specification to map an IEC 61131-3 program to OPC UA XML is provided.

5.1 Specifications

As described in 2.2.3 Data Model OPC UA Information Model follows a layered approach. The two specifications used, belong to the Collaboration Models layer of the Information Model. In Figure 5.1 we can see the OPC UA Information Model layered approach in action. The OPC UA layer defines the basic *Nodes* that each OPC UA Server needs to implement to have a functional empty server. The next layer builds on the previous one. The OPC UA Device Integration layer uses some of the *ObjectType* Nodes from the OPC UA layer to create it's own *ObjectTypes* representing generic *real-world* devices and it's configuration properties in the *AddressSpace* of an empty *device-enabled* OPC UA Server. The OPC UA IEC 61131-3 layer extends the OPC UA DI layer to create it's own set of *ObjectTypes* Nodes representing the IEC 61131-3 software model, described in 3.3.1 Software Model. The Examples layer represents a *real-world* PLC and it's elements as various *ObjectTypes* extended from the OPC UA IEC 61131-3 layer with it's respective *Objects* and *Variables* attached.

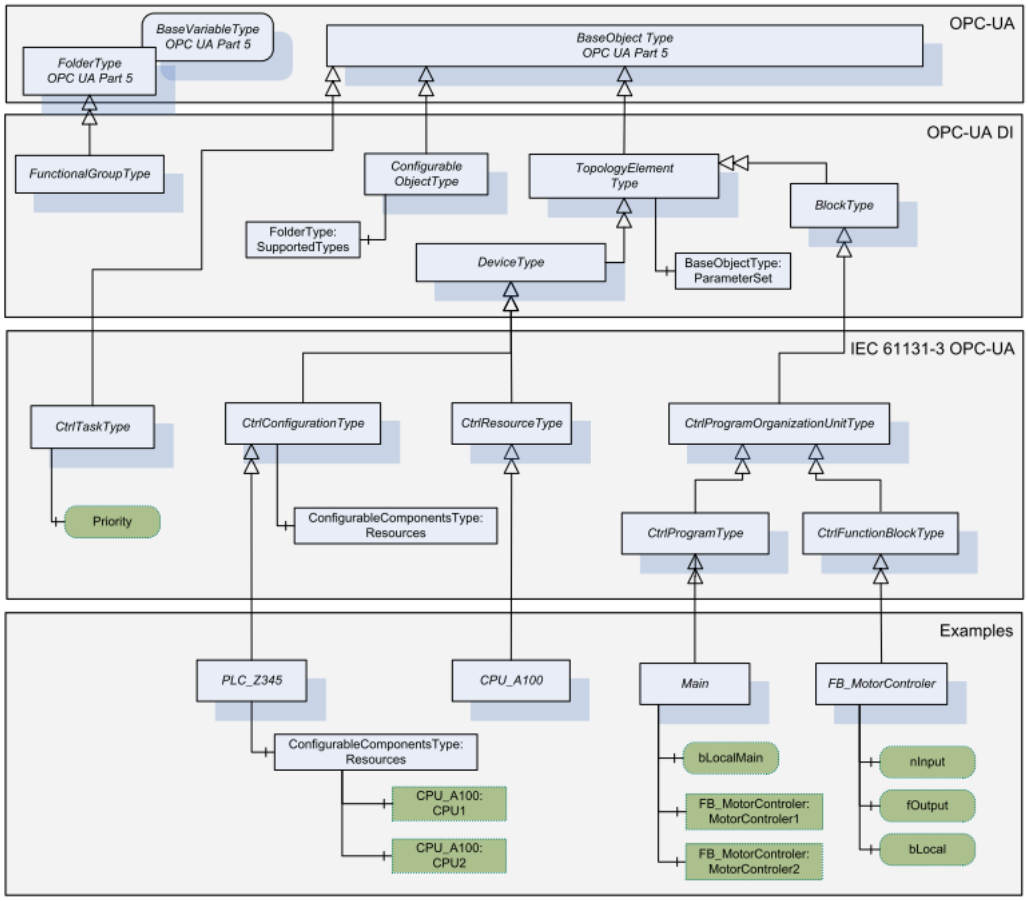

Figure 5.1: OPC UA IEC 61131-3 Model Diagram [11]

5.1.1 OPC UA Information Model for IEC 61131-3

PLCopen and OPC Foundation joined forces to create a specification that mapped IEC 61131-3 architectural model into the OPC UA information model. This fusion of technologies will provide vertical integration for controller systems and facilitate the development of OPC UA servers for IEC 61131-3 compliant devices. The Information model described follows the IEC 61131-3 version 3, from 2013. This specification extends the information model described in OPC UA Device Integration Companion Specification. OPC Foundation provides this information model in XML format, available in their UA-Nodeset/PLCopen github repository [17] [11].

5.1.2 OPC UA Device Integration Information Model

OPC UA DI (Device Integration) is an extension of the OPC UA Information Model and was published as a Companion Specification. It was designed with the intent of creating a general information model for devices. This model would be independent from specific protocols and type of devices. Instead, it would provide a unified model that could be extended by the various manufacturers for their specific products, providing an effortless integration. This information model in XML is available at UA-Nodeset/DI [16] [12].

5.2 Describing the *ObjectTypes*

In this section the most important *ObjectTypes* in the Information Model for IEC 61131-3 and it's respective super-types are described. The respective mapping of this *ObjectTypes* to XML is also described. The XML elements follow the most recent OPC UA XML Scheme provided by OPC Foundation in their UA-Nodeset github repository [80].

In the XML Listings Examples the *namespace 0* represents the OPC UA basic layer . The *namespace 1* represents the OPC UA DI layer, and the *namespace 2* the OPC UA IEC 61131-3. The *namespace 3* will be used for the example in 5.3 Example XML mapping. Only the XML element representing the *Node* being described is represented in each XML Example. The other *Nodes* they reference, are described in the full XML document for each Information Model, all of them available in the OPC Foundation repository [80].

5.2.1 OPC UA Information Model

This specification defines standardized Nodes of the server's *AddressSpace*. This Node types can be sub-typed to create new objects. The three Node types defined in this specification that are used in the OPC UA DI specification - *BaseObjectType*, *BaseVariableType* and *FolderType* - are described.

5.2.1.1 BaseObjectType

Attribute	**Value**				
BrowseName	BaseObjectType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasSubtype	ObjectType	FolderType			
HasSubtype	ObjectType	ConfigurableObjectType			
HasSubtype	ObjectType	TopologyElementType			

Table 5.1: *BaseObjectType* Node [13]

This *ObjectType* Node is used as a base type definition for objects without a proper concrete type defined. The specification advises user's not to use directly this object and instead create more concrete type definitions by sub-typing this *ObjectType*, as was done with the *FolderType*, *ConfigurableObjectType* and *TopologyElementType*.

```
<UAObjectType NodeId="ns=0;i=58" BrowseName="0:BaseObjectType">
  <DisplayName>BaseObjectType</DisplayName>
  <Description>The base type for all object nodes.</Description>
  <References>
    <Reference ReferenceType="HasSubtype">ns=0;i=61</Reference>
  <References/>
```

```
</UAObjectType>
```

Listing 5.1: *BaseObjectType* XML UAObjectType [15]

5.2.1.2 BaseVariableType

Attribute	Value				
BrowseName	BaseVariableType				
IsAbstract	True				
ArraySize	-1				
DataType	BaseDataType				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasSubtype	VariableType	PropertyType			
HasSubtype	ObjectType	BaseDataVariableType			

Table 5.2: *BaseVariableType* Node [13]

This *VariableType* is the abstract base for all other *VariableTypes*. *PropertyType* and *BaseDataVariableType* are the only two sub-types of this Node. If the user wants to create a new *VariableType*, one or the other must be sub-typed, depending if it's *VariableType* will be a *Property* or a *DataVariable*.

```
<UAVariableType NodeId="ns=0;i=62" BrowseName="0:BaseVariableType" IsAbstract="true" ValueRank="−2">
  <DisplayName>BaseVariableType</DisplayName>
  <Description>The abstract base type for all variable nodes.</Description>
  <References>
    <Reference ReferenceType="HasSubtype">ns=0;i=63<Reference/>
    <Reference ReferenceType="HasSubtype">ns=0;i=68<Reference/>
  <References/>
</UAVariableType>
```

Listing 5.2: *BaseVariableType* XML UAVariableType [15]

5.2.1.3 FolderType

Attribute	Value				
BrowseName	FolderType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasTypeDefinition	ObjectType	BaseObjectType			

Table 5.3: *FolderType* Node [13]

This *ObjectType* represents the root Node of a subtree. *Object* Nodes of this type are used to organize the *AddressSpace* into a hierarchy of Nodes based on user's criteria.

```
<UAObjectType NodeId="ns=0;i=61" BrowseName="0:FolderType">
  <DisplayName>FolderType</DisplayName>
  <Description>The type for objects that organize other nodes.</Description>
  <References>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=0;i=58</Reference>
  </References>
</UAObjectType>
```

Listing 5.3: *FolderType* XML UAObjectType [15]

5.2.2 Device Integration Companion Specification

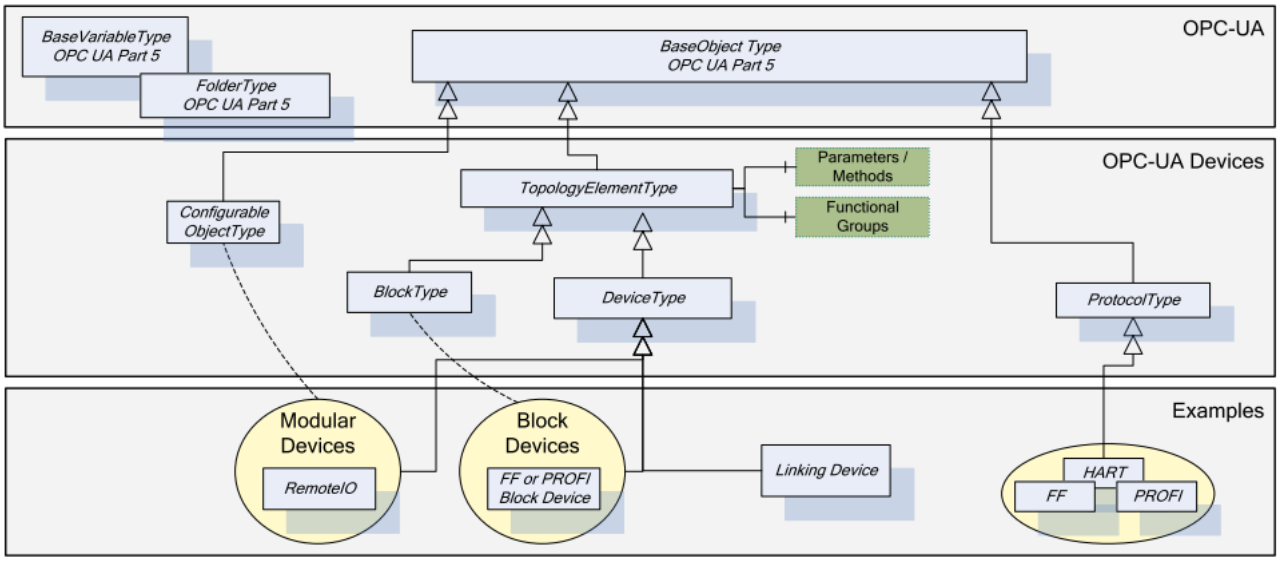

Figure 5.2: OPC UA Devices Model Diagram [12]

This model describes a set of OPC UA *ObjectTypes* to represent the device and it's configuration. We are only interested in the ones that are extended in the Information Model for IEC 61131-3: *TopologyElementType* (and it's subtypes - *DeviceType* and *BlockType*), *FunctionalGroupType* and *ConfigurableObjectType*.

5.2.2.1 TopologyElementType

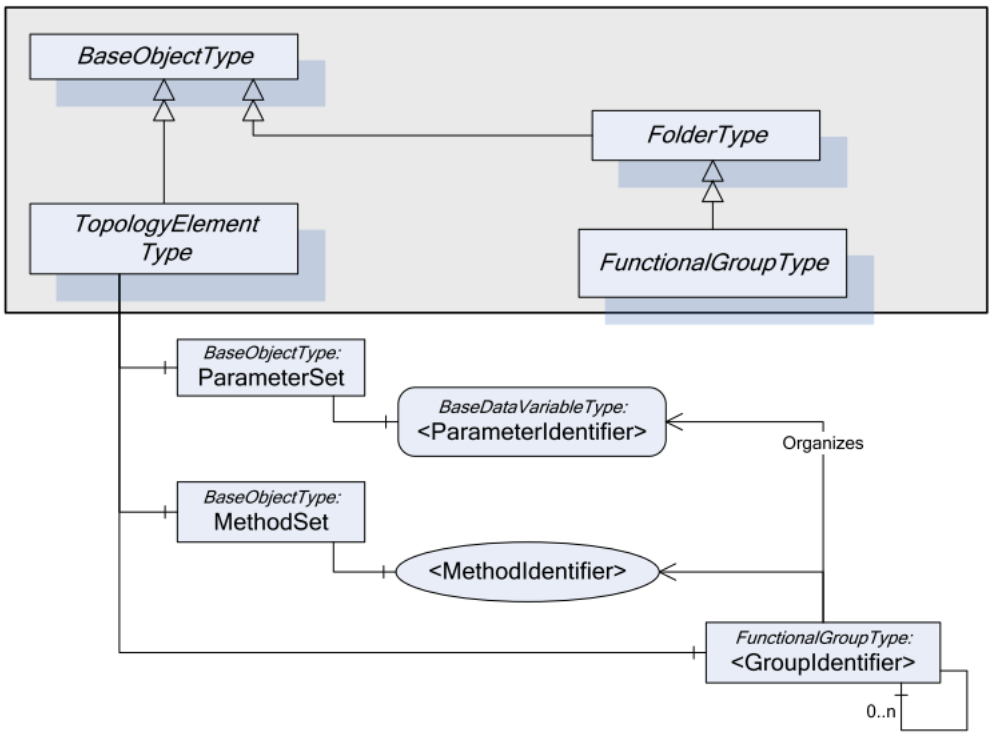

Figure 5.3: OPC UA *TopologyElementType* Diagram [12]

This *ObjectType* is a subtype of the *BaseObjectType*. It defines the basic structure for the configurable elements in a device topology and it has two main components, the *BaseObjectType* ParameterSet and the MethodSet. If the device has associated *Variables* they will be kept has components of the ParameterSet and if it has *Methods* they will be kept has components of the MethodSet. To further structure the *Variables* or *Methods*, FunctionalGroupType objects can be used to organize them based on whatever criteria. As an abstract type this *ObjectType* will have no instances of himself and will have to be sub-typed.

Attribute	**Value**				
BrowseName	TopologyElementType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasComponent	Object	ParameterSet		BaseObjectType	Optional
HasComponent	Object	MethodSet		BaseObjectType	Optional
HasComponent	Object	<GroupIdentifier >		FunctionalGroupType	Optional
HasComponent	Object	Identification		FunctionalGroupType	Optional
HasComponent		Lock			
HasSubtype	ObjectType	DeviceType			
HasSubtype	ObjectType	BlockType			
HasTypeDefinition	ObjectType	BaseObjectType			

Table 5.4: *TopologyElementType* Node [12]

```
<UAObjectType NodeId="ns=1;i=1001" BrowseName="1:TopologyElementType" IsAbstract="true">
  <DisplayName>TopologyElementType</DisplayName>
  <Description>TopologyElementType</Description>
  <References>
    <Reference ReferenceType="HasComponent">ns=1;i=5002</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=5003</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=6567</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=6014</Reference>
    <Reference ReferenceType="HasSubtype">ns=1;i=1002</Reference>
    <Reference ReferenceType="HasSubtype">ns=1;i=1003</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=0;i=58</Reference>
  </References>
</UAObjectType>
```

Listing 5.4: *TopologyElementType* XML UAObjectType [16]

5.2.2.2 DeviceType

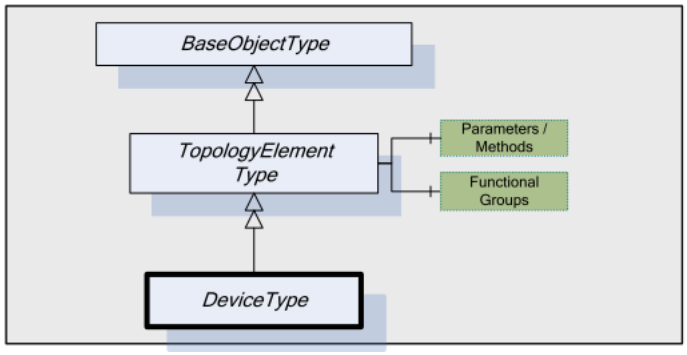

Figure 5.4: OPC UA *DeviceType* Diagram [12]

This *ObjectType* is a sub-type of the *TopologyElementType*. It is also an abstract type and works as a scheme for the Device model. User's will extend this *ObjectType* and create their own

specific *DeviceTypes*. The *Properties* attached to this object provide a way for the object to expose standard device information to the clients.

Attribute	**Value**				
BrowseName	DeviceType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Components* of the *TopologyElementType*					
HasComponent		DeviceTypeImage			
HasComponent		Documentation			
HasComponent		ProtocolSupport			
HasComponent		ImageSet			
HasComponent		<CPIdentifier>			
HasProperty	Variable	SerialNumber	String	PropertyType	Mandatory
HasProperty	Variable	RevisionCounter	Int32	PropertyType	Mandatory
HasProperty	Variable	Manufacturer	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	Model	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	DeviceManual	String	PropertyType	Mandatory
HasProperty	Variable	DeviceRevision	String	PropertyType	Mandatory
HasProperty	Variable	SoftwareRevision	String	PropertyType	Mandatory
HasProperty	Variable	HardwareRevision	String	PropertyType	Mandatory
HasProperty	Variable	DeviceClass	String	PropertyType	
HasProperty	Variable	DeviceHealth	DeviceHealthEnumeration	PropertyType	
HasTypeDefinition	ObjectType	TopologyElementType			

Table 5.5: *DeviceType* Node [12]

```
<UAObjectType NodeId="ns=1;i=1002" BrowseName="1:DeviceType" IsAbstract="true">
  <DisplayName>DeviceType</DisplayName>
  <Description>Defines the basic information components for all configurable elements in a device topology</Description>
  <References>
    <Reference ReferenceType="HasComponent">ns=1;i=6209</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=6211</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=6213</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=6215</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=6571</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6001</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6002</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6003</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6004</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6005</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6006</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6007</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6008</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6470</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6208</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=1;i=1001</Reference>
  </References>
</UAObjectType>
```

Listing 5.5: *DeviceType* XML UAObjectType [16]

5.2.2.3 DeviceSet

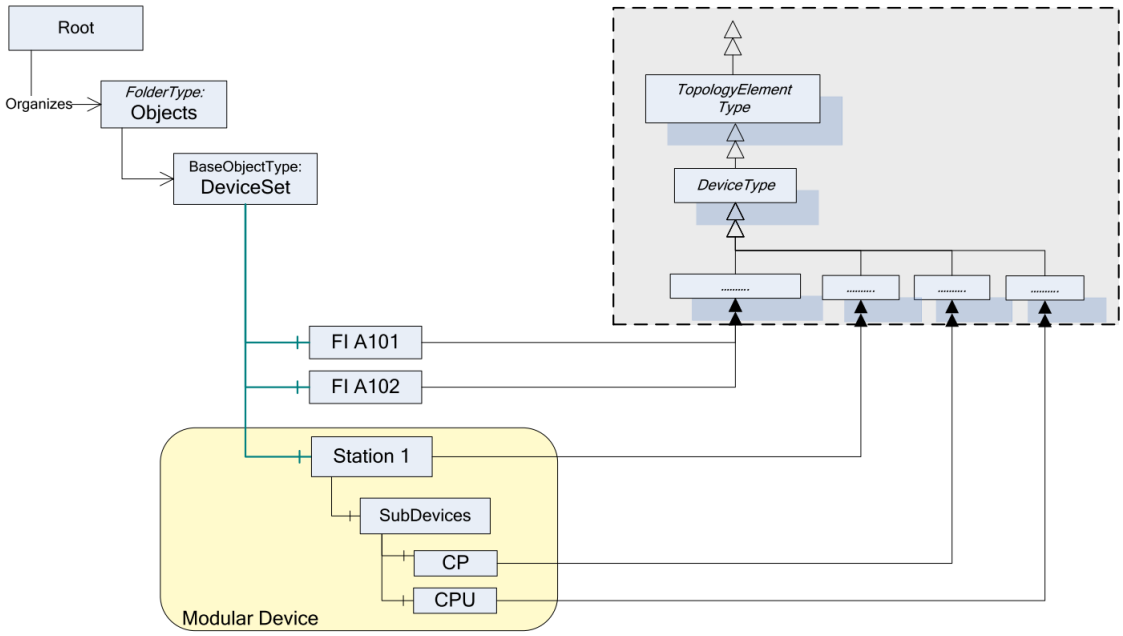

Figure 5.5: OPC UA *DeviceSet* Diagram [12]

All the *Device Objects* must be added to *DeviceSet Object* has *Components*.

```
<UAObject NodeId="ns=1;i=5001" BrowseName="1:DeviceSet">
  <DisplayName>DeviceSet</DisplayName>
  <Description>Contains all instances of devices</Description>
  <References>
    <Reference ReferenceType="Organizes" IsForward="false">ns=0;i=85</Reference>
    <Reference ReferenceType="HasTypeDefinition">ns=0;i=58</Reference>
  </References>
</UAObject>
```

Listing 5.6: *DeviceSet* XML UAObject [16]

5.2.2.4 BlockType

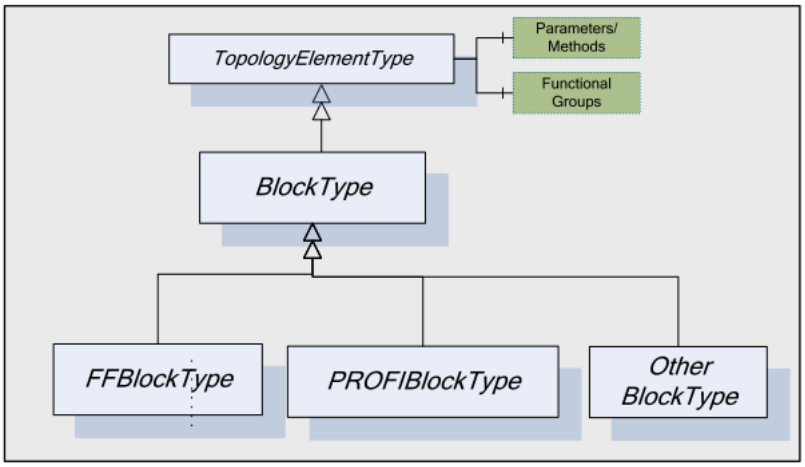

Figure 5.6: OPC UA *BlockType* Diagram [12]

BlockType is an abstract type, extended from the *TopologyElementType* and was designed to provide block-oriented capabilities for FieldDevices. Fieldbus organizations can create their own specific *BlockTypes*. The set of *Properties* attached to this object will expose the set of operations modes that this *Block* supports.

Attribute	**Value**				
BrowseName	BlockType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModRule**
Inherit the *Components* of the *TopologyElementType*					
HasProperty	Variable	RevisionCounter	Int32	PropertyType	Mandatory
HasProperty	Variable	ActualModel	LocalizedText	PropertyType	Optional
HasProperty	Variable	PermittedMode	LocalizedText[]	PropertyType	Optional
HasProperty	Variable	NormalMode	LocalizedText[]	PropertyType	Optional
HasProperty	Variable	TargetMode	LocalizedText[]	PropertyType	Optional
HasSubtype	ObjectType	CtrlProgramOrganizationUnitType			
HasTypeDefinition	ObjectType	TopologyElementType			

Table 5.6: *BlockType* Node [12]

```
<UAObjectType NodeId="ns=1;i=1003" BrowseName="1:BlockType" IsAbstract="true">
  <DisplayName>BlockType</DisplayName>
  <Description>Adds the concept of Blocks needed for block−oriented FieldDevices</Description>
  <References>
    <Reference ReferenceType="HasProperty">ns=1;i=6009</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6010</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6011</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6012</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6013</Reference>
    <Reference ReferenceType="HasSubtype">ns=2;i=1003</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=1;i=1001</Reference>
```

```
  </References>
</UAObjectType>
```

Listing 5.7: *BlockType* XML UAObjectType [16]

5.2.2.5 ConfigurableObjectType

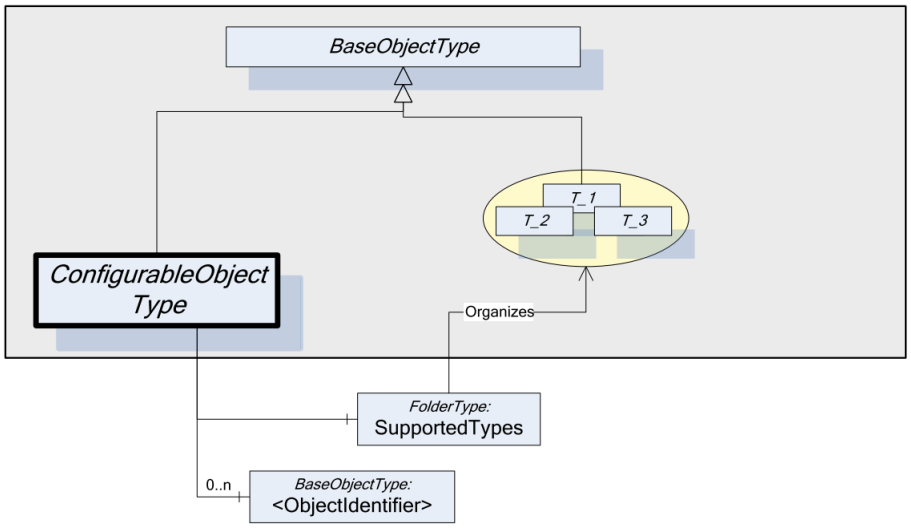

Figure 5.7: OPC UA *ConfigurableObjectType* Diagram [12]

Attribute	**Value**				
BrowseName	ConfigurableObjectType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModRule**
HasComponent	Object	SupportedTypes		FolderType	Mandatory
HasComponent	Object	<ObjectIdentifier>		BaseObjectType	Optional
HasTypeDefinition	ObjectType	BaseObjectType			

Table 5.7: *ConfigurableObjectType* Node [12]

This *ObjectType* is used to provide configuration capability to *Object* instances and implements the Configurable Component pattern, described in section 5.11.1 of this specification. The *SupportedTypes* folder maintains the *ObjectTypes* that can be instantiated in this configurable *Object*.

```
<UAObjectType NodeId="ns=1;i=1004" BrowseName="1:ConfigurableObjectType">
  <DisplayName>ConfigurableObjectType</DisplayName>
  <Description>Defines a general pattern to expose and configure modular components</Description>
  <References>
    <Reference ReferenceType="HasComponent">ns=1;i=5004</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=6026</Reference>
```

```
    <Reference ReferenceType="HasSubtype" IsForward="false">i=58</Reference>
  </References>
</UAObjectType>
```

Listing 5.8: *ConfigurableObjectType* XML UAObjectType [16]

5.2.2.6 FunctionalGroupType

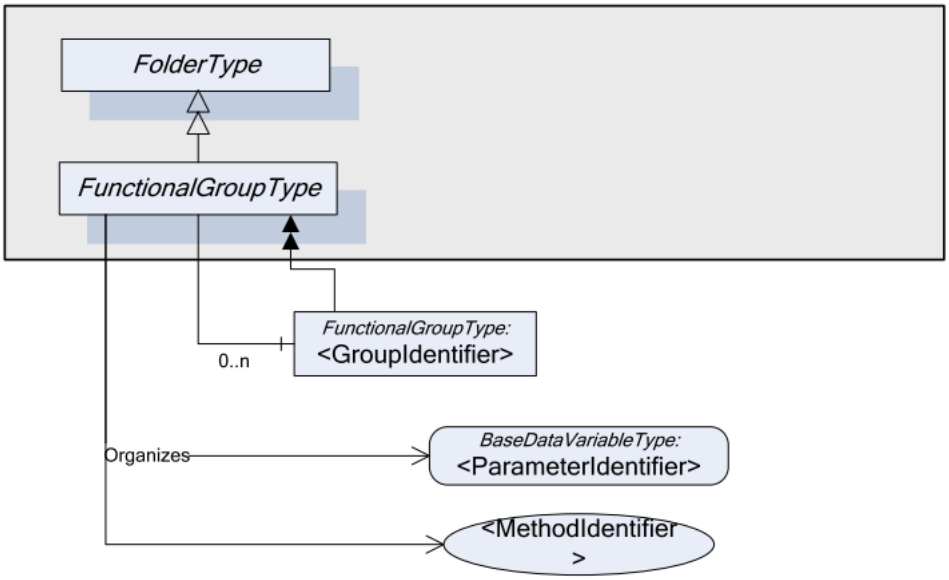

Figure 5.8: OPC UA *FunctionalGroupType* Diagram [12]

Attribute	Value				
BrowseName	FunctionalGroupType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModRule**
HasComponent	Object	<GroupIdentifier>		FunctionalGroupType	Optional
Organizes	Variable	<ParameterIdentifier>		BaseDataVariableType	Optional
Organizes	Variable	<MethodIdentifier>			Optional
HasComponent	Variable	UIElement		UIElementType	Optional
HasTypeDefinition	ObjectType	FolderType			

Table 5.8: *FunctionalGroupType* Node [12]

This *ObjectType* is a sub-type of the *FolderType* and is used to organize the *Parameters* and *Methods* from the *ParameterSet* and *MethodSet* based on some criteria.

```
<UAObjectType NodeId="ns=1;i=1005" BrowseName="1:FunctionalGroupType">
<DisplayName>FunctionalGroupType</DisplayName>
  <Description>Used to organize the Parameters and Methods</Description>
  <References>
```

```
    <Reference ReferenceType="HasComponent">ns=1;i=6027</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=6028</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=6029</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=6243</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=0;i=61</Reference>
  </References>
</UAObjectType>
```

Listing 5.9: *FunctionalGroupType* XML UAObjectType [16]

5.2.3 Information Model for IEC 61131-3

5.2.3.1 CtrlConfigurationType

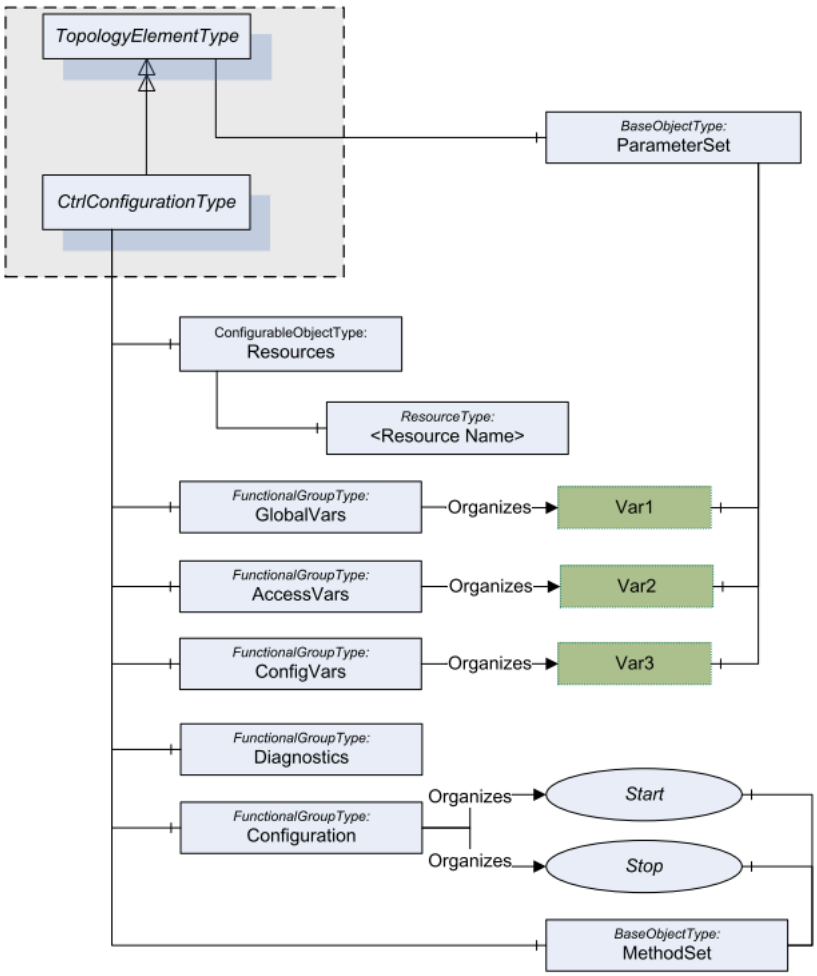

Figure 5.9: OPC UA *CtrlConfigurationType* Diagram [11]

The *CtrlConfigurationType* represents the IEC 61131-3 *Configuration*. It is a subtype of the *TopologyElementType* and can be instantiated directly because it is not abstract. The specification

recommends the manufacturers to implement their own types by sub-typing the *CtrlConfigurationType*.

Attribute	**Value**				
BrowseName	CtrlConfigurationType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Components* of the *TopologyElementType*					
HasComponent	Object	MethodSet		BaseObjectType	Optional
HasComponent	Object	Resources		ConfigurableObjectType	Mandatory
HasComponent	Object	GlobalVars		FunctionalGroupType	Optional
HasComponent	Object	AccessVars		FunctionalGroupType	Optional
HasComponent	Object	ConfigVars		FunctionalGroupType	Optional
HasComponent	Object	Configuration		FunctionalGroupType	Optional
HasComponent	Object	Status		FunctionalGroupType	Optional
HasTypeDefinition	ObjectType	TopologyElementType			

Table 5.9: *CtrlConfigurationType* Node [11]

In a *CtrlConfiguration Object* the *Resources Object* is used to organize it's *CtrlResources*. The *GlobalVars* is used to aggregate the *CtrlVariables* declared as VAR_GLOBAL, the *AccessVars* the ones declared as VAR_ACCESS and *ConfigVars* the ones declared as VAR_CONFIG. The *Status Object* contains diagnostic and status information.

```
<UAObjectType NodeId="ns=2;i=1001" BrowseName="2:CtrlConfigurationType">
  <DisplayName>CtrlConfigurationType</DisplayName>
  <References>
    <Reference ReferenceType="HasComponent">ns=2;i=5002</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5004</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5006</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5007</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5008</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5009</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5010</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=1;i=1002</Reference>
  </References>
</UAObjectType>
```

Listing 5.10: *CtrlConfigurationType* XML UAObjectType [17]

5.2.3.2 CtrlResourceType

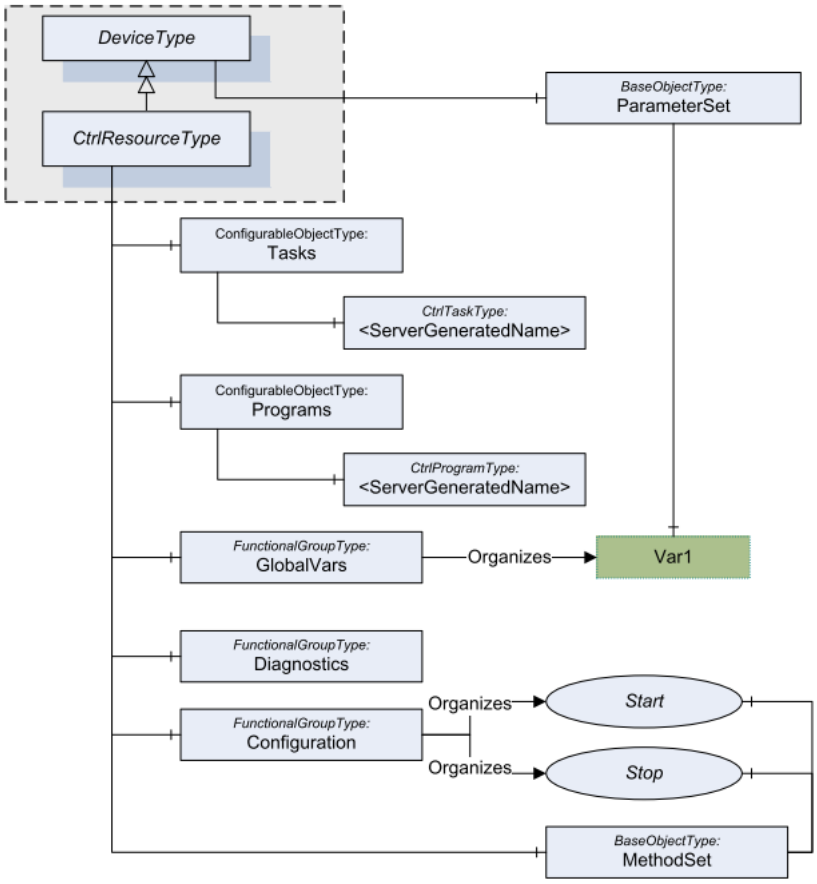

Figure 5.10: OPC UA *CtrlResourceType* Diagram [11]

This *ObjectType* is a sub-type of the *DeviceType* and represents the IEC 61131-3 *Resource*. It is a concrete type and can be instantiated directly. As in the *CtrlConfigurationType* the OPC Foundation recommends vendors to create their own *CtrlResourceTypes*.

Attribute	**Value**				
BrowseName	CtrlResourceType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Components* of the *DeviceType*					
HasComponent	Object	MethodSet		BaseObjectType	Optional
HasComponent	Object	Tasks		ConfigurableObjectType	Mandatory
HasComponent	Object	Programs		ConfigurableObjectType	Mandatory
HasComponent	Object	GlobalVars		FunctionalGroupType	Optional
HasComponent	Object	Configuration		FunctionalGroupType	Optional
HasComponent	Object	Status		FunctionalGroupType	Optional
HasTypeDefinition	ObjectType	DeviceType			

Table 5.10: *CtrlResourceType* Node [11]

The *Tasks Object* is used to group *CtrlTasks* that are part of the *CtrlResource Object*. The *Programs* groups the *CtrlPrograms* that are part of the *CtrlResource*. As in the *CtrlConfigurationType* the *GlobalVars Object* contains the *CtrlVariables* declared as global and the *Status* contains diagnostic information.

```
<UAObjectType NodeId="ns=2;i=1002" BrowseName="2:CtrlResourceType">
  <DisplayName>CtrlResourceType</DisplayName>
  <References>
    <Reference ReferenceType="HasComponent">ns=2;i=5012</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5014</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5016</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5018</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5019</Reference>
    <Reference ReferenceType="HasComponent">ns=2;i=5020</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=1;i=1002</Reference>
  </References>
</UAObjectType>
```

Listing 5.11: *CtrlResourceType* XML UAObjectType [17]

5.2.3.3 CtrlTaskType

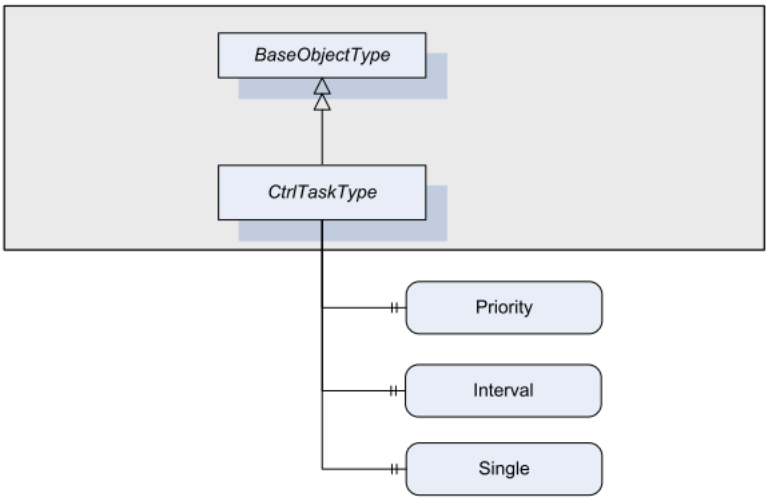

Figure 5.11: OPC UA *CtrlTaskType* Diagram [11]

Attribute	Value				
BrowseName	CtrlTaskType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	Priority	UInt32	PropertyType	Mandatory
HasProperty	Variable	Interval	String	PropertyType	Optional
HasProperty	Variable	Single	String	PropertyType	Optional

Table 5.11: *CtrlTaskType* Node [11]

The *CtrlTaskType* is the *ObjectType* defining the IEC 61131-3 Task. It has three *Component Properties* each defining the IEC 61131-3 Task options of the associated Run-time POU. The *Priority* indicates the scheduling priority associated with the POU, the *Interval* the periodical scheduling at the specified interval and *Single* indicates the scheduling of the POU at each rising edge.

```
<UAObjectType NodeId="ns=2;i=1006" BrowseName="2:CtrlTaskType">
  <DisplayName>CtrlTaskType</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty">ns=2;i=6004</Reference>
    <Reference ReferenceType="HasProperty">ns=2;i=6005</Reference>
    <Reference ReferenceType="HasProperty">ns=2;i=6006</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=0;i=58</Reference>
  </References>
</UAObjectType>
```

Listing 5.12: *CtrlTaskType* XML UAObjectType [17]

5.2.3.4 CtrlProgramOrganizationUnitType

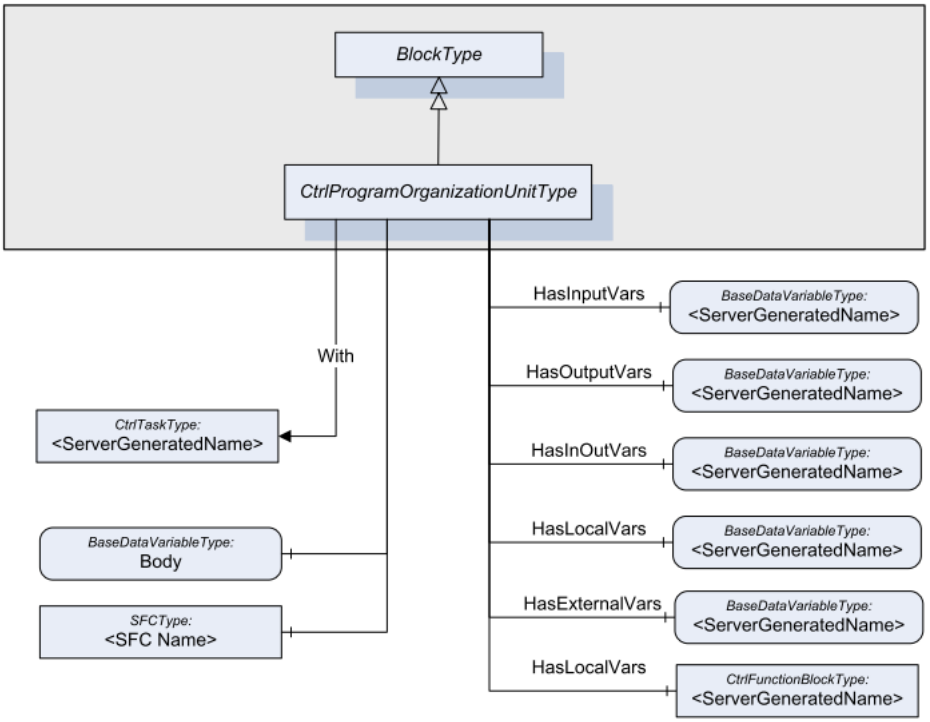

Figure 5.12: OPC UA *CtrlProgramOrganizationUnitType* Diagram [11]

This *ObjecType* defines the POU representation in OPC UA and is a sub-type of the *BlockType* defined in the OPC UA DI Information Model. It is an abstract type and will be extended by the *CtrlProgramType* and the *CtrlFunctionBlockType* to represent the IEC 61131-3 Program and Function Block respectively. The Task associated with the POU instance is defined by using the *With* Reference, defined in section 4.7.7 of this specification, pointing to the *CtrlTask* object that represents that Task.

Attribute	Value					
BrowseName	CtrlProgramOrganizationUnitType					
IsAbstract	True					
References	**Cardinality**	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModRule**
Inherit the *Components* of the *BlockType*						
With	0 - N	Object	<Task Name >		CtrlTaskType	Optional
HasInputVar	0 - N	Variable	<Var Name >	BaseDataType	BaseDataVariableType	Optional
HasOutputVar	0 - N	Variable	<Var Name >	BaseDataType	BaseDataVariableType	Optional
HasInOutVar	0 - N	Variable	<Var Name >	BaseDataType	BaseDataVariableType	Optional
HasLocalVar	0 - N	Variable	<Var Name >	BaseDataType	BaseDataVariableType	Optional
HasExternalVar	0 - N	Variable	<Var Name >	BaseDataType	BaseDataVariableType	Optional
HasLocalVar	0 - N	Object	<Block Name >		CtrlFunctionBlockType	Optional
HasComponent	0 - N	Object	<SFC Name >		SFCType	Optional
HasComponent		Variable	Body	XmlElement	BaseDataVariableType	Optional
HasSubtype		ObjectType	CtrlProgramType			
HasSubtype		ObjectType	CtrlFunctionBlockType			
HasTypeDefinition		ObjectType	BlockType			

Table 5.12: *CtrlProgramOrganizationUnitType* Node [11]

The variables declared in the POU are mapped to OPC UA *Variables* and are referenced in the *CtrlProgramOrganizationUnit* using different subtypes of the *HasComponent* Reference. All of the subtypes are defined in section 4.7 of this specification. Variables declared with te VAR_INPUT keyword are referenced using the *HasInputVar* (section 4.7.2), the ones declared with VAR_OUTPUT use the *HasOutputVar* reference (section 4.7.3), the ones with VAR_IN_OUT to the *HasInOutVar* (section 4.7.4), the VAR_EXTERNAL to *HasExternalVar* (section 4.7.6) and finnaly, local variables, simply declared with just VAR are referenced using *HasLocalVar* (section 4.7.5).

```
<UAObjectType NodeId="ns=2;i=1003" BrowseName="2:CtrlProgramOrganizationUnitType" IsAbstract="true">
  <DisplayName>CtrlProgramOrganizationUnitType</DisplayName>
  <References>
    <Reference ReferenceType="HasComponent">ns=2;i=6001</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=1;i=1003</Reference>
  </References>
</UAObjectType>
```

Listing 5.13: *CtrlProgramOrganizationUnitType* XML UAObjectType [17]

5.2.3.5 CtrlProgramType

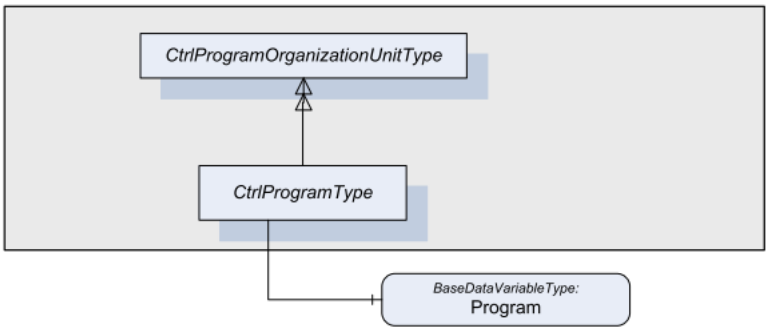

Figure 5.13: OPC UA *CtrlProgramType* Diagram [11]

Attribute	Value				
BrowseName	CtrlProgramType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* and *Components* of the CtrlProgramOrganizationUnitType					
HasComponent	Variable	Program	Structure	BaseDataVariableType	Optional
HasTypeDefinition	ObjectType	CtrlProgramOrganizationUnitType			

Table 5.13: *CtrlProgramType* Node [11]

The *CtrlProgramType* is an abstract type that represents the Program POU. The Program *Variable* that is attached to it can be used to save the complete program declaration in a complex Variable.

```
<UAObjectType NodeId="ns=2;i=1004" BrowseName="2:CtrlProgramType" IsAbstract="true">
  <DisplayName>CtrlProgramType</DisplayName>
  <References>
    <Reference ReferenceType="HasComponent">ns=2;i=6002</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=2;i=1003</Reference>
  </References>
</UAObjectType>
```

Listing 5.14: *CtrlProgramType* XML UAObjectType [17]

5.2.3.6 CtrlFunctionBlockType

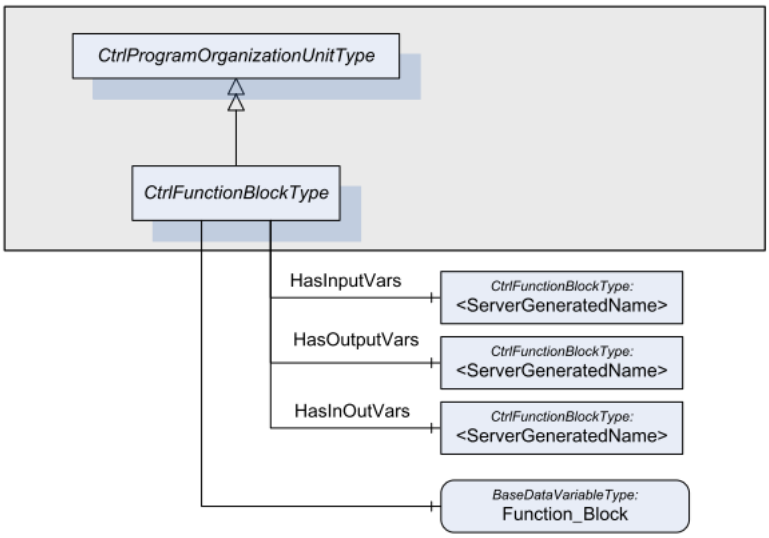

Figure 5.14: OPC UA *CtrlFunctionBlockType* Diagram [11]

Attribute	**Value**					
BrowseName	CtrlFunctionBlockType					
IsAbstract	True					
References	**Cardinality**	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* and *Components* of the *CtrlProgramOrganizationUnitType*						
HasComponent	1	Variable	FunctionBlock		BaseDataVariableType	Optional
HasInputVar	0 - N	Variable	<Var Name >	BaseDataType	BaseDataVariableType	Optional
HasOutputVar	0 - N	Variable	<Var Name >	BaseDataType	BaseDataVariableType	Optional
HasInOutVar	0 - N	Variable	<Var Name >	BaseDataType	BaseDataVariableType	Optional

Table 5.14: *CtrlFunctionBlockType* Node [11]

This *ObjectType* extends the *CtrlProgramOrganizationUnitType* and is also an abstract type. It will need to be sub-typed for each specific case. Like the *CtrlProgramType* is has an attached variable, the FunctionBlock variable that is used to also save the complete Function_Block declaration in a complex Variable.

```
<UAObjectType NodeId="ns=2;i=1005" BrowseName="2:CtrlFunctionBlockType" IsAbstract="true">
  <DisplayName>CtrlFunctionBlockType</DisplayName>
  <References>
    <Reference ReferenceType="HasComponent">ns=2;i=6003</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=2;i=1003</Reference>
  </References>
</UAObjectType>
```

Listing 5.15: *CtrlFunctionBlockType* XML UAObjectType [17]

5.2.3.7 SFCType

Attribute	Value				
BrowseName	SFCType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasTypeDefinition	ObjectType	BaseObjectType			

Table 5.15: *SFCType* Node [11]

```
<UAObjectType NodeId="ns=2;i=1007" BrowseName="2:SFCType">
  <DisplayName>SFCType</DisplayName>
  <References>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=0;i=58</Reference>
  </References>
</UAObjectType>
```

Listing 5.16: *SFCType* XML UAObjectType [17]

5.2.3.8 Data Types

Data Type mappings are described in section 5.2 of this specification. The mapping of the elementary IEC 61131-3 data types to the OPC UA data types is specified in section 5.2.1, the mapping of generic data types in section 5.2.2 and the mapping of derived data types is extensively detailed in section 5.2.3.

5.3 Example XML mapping

This example is not part of the OPC UA IEC 61131-3 specification and was created to explain the basic concepts of mapping a IEC 61131-3 complete PLC program to OPC UA.

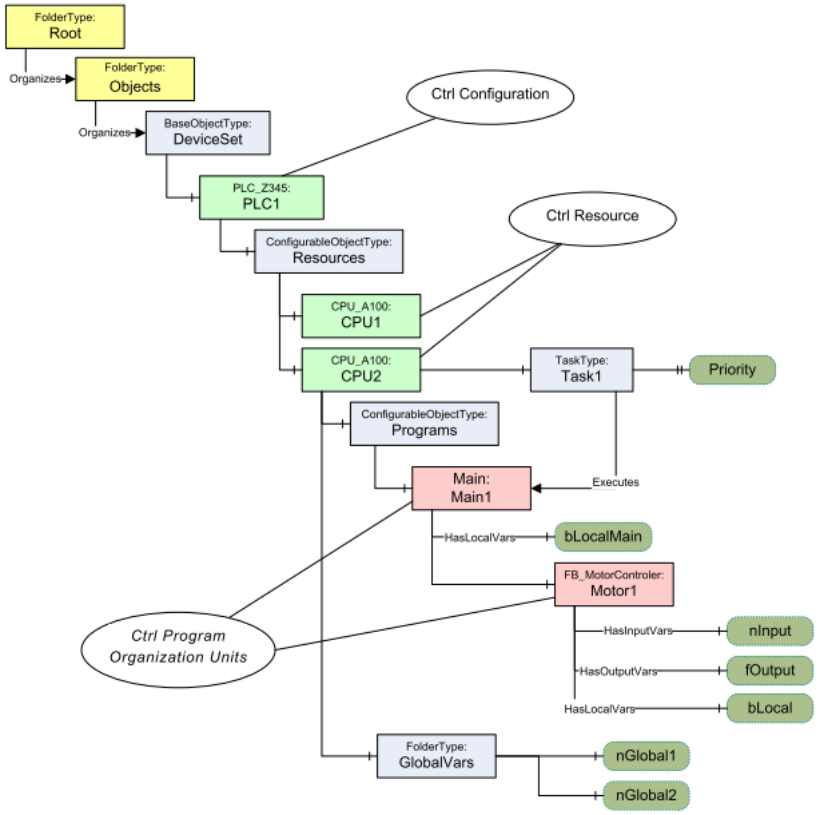

Figure 5.15: OPC UA *AddressSpace* Structure Diagram - 6.1 [11]

The IEC 61131-3 code is a reverse mapping of a Diagram explaining the *AddressSpace* structure in section 6.1 of the specification and is very simplistic, it was made just to give a brief example. The code is mapped to OPC UA XML Elements using the XML Scheme notation defined for OPC UA and only the most important functionality is described.

```
FUNCTION_BLOCK FB_MotorController
  VAR_INPUT
    nInput: BOOL;
  END_VAR
  VAR_OUTPUT
    fOutput: BOOL;
  END_VAR
  VAR
    bLocal: BOOL;
  END_VAR

  (* BODY *)
END_FUNCTION_BLOCK

PROGRAM Main
  VAR
    bLocalMain: BOOL;
    Motor1: FB_MotorController;
  END_VAR

  (* BODY *)
END_PROGRAM

CONFIGURATION PLC_Z345
  RESOURCE CPU_1 ON CPU_A100
    VAR_GLOBAL
      nGlobal1: BOOL;
      nGlobal2: BOOL;
    END_VAR
    TASK task1(INTERVAL := T#5ms,PRIORITY := 0);
    PROGRAM Main1 WITH task1: Main;
  END_RESOURCE

  RESOURCE CPU_2 ON CPU_A100
    VAR_GLOBAL
      nGlobal1: BOOL;
      nGlobal2: BOOL;
    END_VAR
    TASK task2(INTERVAL := T#5ms,PRIORITY := 0);
    PROGRAM Main1 WITH task2: Main;
  END_RESOURCE
END_CONFIGURATION
```

Listing 5.17: Example IEC 61131-3 code based on OPC UA Diagram

5.3.1 Mapping

5.3.1.1 Function Block

```
FUNCTION_BLOCK FB_MotorController
  VAR_INPUT
    nInput: BOOL;
  END_VAR
  VAR_OUTPUT
    fOutput: BOOL;
  END_VAR
  VAR
    bLocal: BOOL;
  END_VAR

  (* BODY *)
END_FUNCTION_BLOCK
```

Listing 5.18: Example IEC 61131-3 *Function Block* code based on OPC UA Diagram

To map a IEC 61131-3 Function Block POU to OPC UA we'll use the OPC UA CtrlConfigurationType. This ObjectType is abstract, so for each different Function Block type declared in our IEC 61131-3 code we'll create a subtype of the CtrlFunctionBlockType and add the specified variables.

The "HasSubtype" reference points to the namespace and id of the CtrlFunctionBlockType node. The other references map the Function Block variables.

```
<UAObjectType NodeId="ns=3;i=1001" BrowseName="3:FB_MotorController">  <DisplayName>FB_MotorController</DisplayName
  <References>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=2;i=1005</Reference>
        <Reference ReferenceType="HasInputVars">ns=3;i=6001</Reference>
        <Reference ReferenceType="HasOutputVars">ns=3;i=6002</Reference>
        <Reference ReferenceType="HasLocalVars">ns=3;i=6003</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=6004</Reference>
  </References>
</UAObjectType>
```

Listing 5.19: *Function Block* mapping to XML UAObjectType

We need to map each variable declared in the Function Block to an OPC UA Variable.

```
<UAVariable NodeId="ns=3;i=6001" BrowseName="3:nInput" DataType="Boolean">
  <DisplayName>nInput</DisplayName>
  <References>
```

```
    <Reference ReferenceType="HasInputVars" IsForward="false">ns=3;i=1001</Reference>
  </References>
</UAVariable>

<UAVariable NodeId="ns=3;i=6002" BrowseName="3:fOutput" DataType="Boolean">
  <DisplayName>fOutput</DisplayName>
  <References>
    <Reference ReferenceType="HasOutputVars" IsForward="false">ns=3;i=1001</Reference>
  </References>
</UAVariable>

<UAVariable NodeId="ns=3;i=6003" BrowseName="3:bLocal" DataType="Boolean">
  <DisplayName>bLocal</DisplayName>
  <References>
        <Reference ReferenceType="HasLocalVars" IsForward="false">ns=3;i=1001</Reference>
  </References>
</UAVariable>
```

Listing 5.20: Function Block declared *Variables* mapping to XML UAVariable

The complete data of the ST declared Function_Block should be saved as a complex variable in the FunctionBlock *Variable*.

```
<UAVariable NodeId="ns=3;i=6004" BrowseName="3:FunctionBlock" ParentNodeId="ns=3;i=1001" DataType="Structure">
  <DisplayName>FunctionBlock</DisplayName>
  <References>
    <Reference ReferenceType="HasTypeDefinition">ns=0;i=63</Reference>
    <Reference ReferenceType="HasModellingRule">ns=0;i=80</Reference>
    <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=1001</Reference>
  </References>
  <Value>
        <Structure></Structure>
  </Value>
</UAVariable>
```

Listing 5.21: *Variable* mapping to XML UAVariable

5.3.1.2 Program

```
PROGRAM Main
  VAR
    bLocalMain: BOOL;
    Motor1: FB_MotorController;
  END_VAR

  (* BODY *)
END_PROGRAM
```

Listing 5.22: Example IEC 61131-3 Program code based on OPC UA Diagram

The Program POU is mapped to OPC UA *CtrlProgramType*. This is also an abstract class, so we'll need to create the Main ObjectType by subtyping the CtrlProgramType.

```
<UAObjectType NodeId="ns=3;i=1002" BrowseName="3:Main">
  <DisplayName>Main</DisplayName>
  <References>
        <Reference ReferenceType="HasSubtype" IsForward="false">ns=2;i=1004</Reference>
        <Reference ReferenceType="HasLocalVars">ns=3;i=6005</Reference>
        <Reference ReferenceType="HasLocalVars">ns=3;i=5001</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=6006</Reference>
  </References>
</UAObjectType>
```

Listing 5.23: *Program* mapping to XML UAObjectType

The local variable bLocalMain will be mapped to an OPC UA Variable with an equivalent DataType and the local Function Block variable Motor1, that is of type FB_MotorController will be mapped to a OPC UA Object with a "HasTypeDefinition" reference to the FB_MotorController ObjectType defined earlier.

```
<UAVariable NodeId="ns=3;i=6005" BrowseName="3:bLocalMain" DataType="Boolean" ParentNodeId="ns=3;i=1002">
  <DisplayName>bLocalMain</DisplayName>
  <References>
        <Reference ReferenceType="HasLocalVars" IsForward="false">ns=3;i=1002</Reference>
  </References>
</UAVariable>

<UAObject NodeId="ns=3;i=5001" BrowseName="3:Motor1" ParentNodeId="ns=3;i=1002">
  <DisplayName>Motor1</DisplayName>
  <References>
        <Reference ReferenceType="HasLocalVars" IsForward="false">ns=3;i=1002</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=3;i=1001</Reference>
  </References>
```

```
</UAObject>
```

Listing 5.24: Mapping to XML of the declared Variables in the Program code

The complete data of the ST declared Program should be saved as a complex variable in the Program *Variable*.

```
<UAVariable NodeId="ns=3;i=6006" BrowseName="3:Program" ParentNodeId="ns=3;i=1002" DataType="Structure">
  <DisplayName>Program</DisplayName>
  <References>
    <Reference ReferenceType="HasTypeDefinition">ns=0;i=63</Reference>
    <Reference ReferenceType="HasModellingRule">ns=0;i=80</Reference>
    <Reference ReferenceType="HasComponent" IsForward="false">ns3;i=1002</Reference>
  </References>
</UAVariable>
```

Listing 5.25: Variable mapping to XML UAVariable

5.3.1.3 Configuration

```
CONFIGURATION PLC_Z345
 RESOURCE CPU_1 ON CPU_A100
   VAR_GLOBAL
     nGlobal1: BOOL;
     nGlobal2: BOOL;
   END_VAR
   TASK task2(INTERVAL := T#5ms,PRIORITY := 0);
   PROGRAM Main1 WITH task1: Main;
 END_RESOURCE

 RESOURCE CPU_2 ON CPU_A100
   VAR_GLOBAL
     nGlobal1: BOOL;
     nGlobal2: BOOL;
   END_VAR
   TASK task1(INTERVAL := T#5ms,PRIORITY := 0);
   PROGRAM Main1 WITH task2: Main;
  END_RESOURCE
END_CONFIGURATION
```

Listing 5.26: Example IEC 61131-3 Configuration ST code based on OPC UA Diagram

IEC 61131-3 Configurations are mapped to OPC UA *CtrlConfigurationType* objects. *CtrlConfigurationType* is not abstract, you can map your configuration directly to an OPC UA Object

instance that extends *CtrlConfigurationType*. The *CtrlConfiguration* instances are added has components of the *DeviceSet Object* specified in OPC UA DI.

```
<UAObject NodeId="ns=3;i=5002" BrowseName="3:PLC1" ParentNodeId="ns=1;i=5001">
  <DisplayName>PLC1</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=5001</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=2;i=1001</Reference>
  </References>
</UAObject>
```

Listing 5.27: Configuration mapping to XML UAObject

Or you can define an new *ObjectType* that subtypes the *CtrlConfigurationType* and instantiate a new object.

```
<UAObjectType NodeId="ns=3;i=1003" BrowseName="3:PLC_Z345">
  <DisplayName>PLC_Z345</DisplayName>
  <References>
        <Reference ReferenceType="HasSubtype" IsForward="false">ns=2;i=1001</Reference>
  </References>
</UAObjectType>

<UAObject NodeId="ns=3;i=5002" BrowseName="3:PLC1" ParentNodeId="ns=1;i=5001">
  <DisplayName>PLC1</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=5001</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=3;i=1003</Reference>
  </References>
</UAObject>
```

Listing 5.28: *Configuration* mapping to XML UAObjectType and instantiation using UAObject

If not specified the *Component Objects* of the *CtrlConfigurationType* will be automatically instantiated by the server. Because we need to add *CtrlResource Objects* to our *CtrlConfiguration*, we'll need to implement the *Resources ConfigurableObjectType* in our *CtrlConfiguration*.

```
<UAObject NodeId="ns=3;i=5003" BrowseName="3:Resources" ParentNodeId="ns=3;i=5002">
  <DisplayName>Resources</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5002</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1004</Reference>
        <Reference ReferenceType="HasModellingRule">ns=0;i=78</Reference>
  </References>
</UAObject>
```

Listing 5.29: CtrlConfiguration Resources Object mapping to XML UAObject

5.3.1.4 Resource

```
RESOURCE CPU_1 ON CPU_A100
  VAR_GLOBAL
    nGlobal1: BOOL;
    nGlobal2: BOOL;
  END_VAR
  TASK task1(INTERVAL := T#5ms,PRIORITY := 0);
  PROGRAM Main1 WITH task1: Main;
END_RESOURCE
```

Listing 5.30: Example IEC 61131-3 *Resource* based on OPC UA Diagram

IEC 61131-3 Resources are mapped to OPC UA *CtrlResourceType* objects. *CtrlResourceType* is a concrete type, you can map your configuration directly to a OPC UA *CtrlResourceType* Object. This *CtrlResource Object* will be a *Component* of the *Resources Object* defined above.

```
<UAObject NodeId="ns=3;i=5004" BrowseName="3:CPU_1" ParentNodeId="ns=3;i=5003">
  <DisplayName>CPU_1</DisplayName>
  <References>
    <Reference ReferenceType="HasComponent">ns=3;i=5006</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=5007</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=5008</Reference>
      <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5003</Reference>
      <Reference ReferenceType="HasTypeDefinition">ns=2;i=1002</Reference>
  </References>
</UAObject>
```

Listing 5.31: *Resource* mapping to XML UAObject

Or you can define an new ObjectType that subtypes the CtrlResourceType and instantiate a new object.

```
<UAObjectType NodeId="ns=3;i=1004" BrowseName="3:CPU_A100">
  <DisplayName>CPU_A100</DisplayName>
  <References>
    <Reference ReferenceType="HasComponent">ns=3;i=5006</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=5007</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=5008</Reference>
      <Reference ReferenceType="HasSubtype" IsForward="false">ns=2;i=1002</Reference>
  </References>
</UAObject>

<UAObject NodeId="ns=3;i=5005" BrowseName="3:CPU_1" ParentNodeId="ns=3;i=5003>
  <DisplayName>CPU_1</DisplayName>
  <References>
```

```
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5003</Reference>
      <Reference ReferenceType="HasTypeDefinition">ns=3;i=1004</Reference>
    </References>
</UAObject>
```

Listing 5.32: *Resource* mapping to XML UAObjectType and instantiation using UAObject

Add the Objects specified in *CtrlResourceType* that you'll need to use.

```
<UAObject NodeId="ns=3;i=5006" BrowseName="3:Tasks" ParentNodeId="ns=3;i=5005">
  <DisplayName>Tasks</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5005</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1004</Reference>
        <Reference ReferenceType="HasModellingRule">ns=0;i=78</Reference>
  </References>
</UAObject>

<UAObject NodeId="ns=3;i=5007" BrowseName="3:Programs" ParentNodeId="ns=3;i=5005">
  <DisplayName>Programs</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5005</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1004</Reference>
        <Reference ReferenceType="HasModellingRule">ns=0;i=78</Reference>
  </References>
</UAObject>

<UAObject NodeId="ns=3;i=5008" BrowseName="3:GlobalVars" ParentNodeId="ns=3;i=5005">
  <DisplayName>GlobalVars</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5005</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1005</Reference>
        <Reference ReferenceType="HasModellingRule">ns=0;i=78</Reference>
  </References>
</UAObject>
```

Listing 5.33: Resource Component Objects mapping to XML UAObject

5.3.1.5 Resource - *GlobalVars*

```
RESOURCE CPU_1 ON CPU_A100
   VAR_GLOBAL
     nGlobal1: BOOL;
     nGlobal2: BOOL;
   END_VAR
   ...
 END_RESOURCE
```

Listing 5.34: Example IEC 61131-3 *Resource* based on OPC UA Diagram

For each of the *GlobalVariables* declared in this *Resource* a *Variable* Node will be specified. To describe them as *GlobalVariables* this *Variables* will be *Components* of the *CtrlResource GlobalVars Object*. This *Object* is instantiated by default because it's defined in the *ObjectType* but because we want to add the variables we need to implement it.

```
<UAObject NodeId="ns=3;i=5008" BrowseName="3:GlobalVars" ParentNodeId="ns=3;i=5005">
  <DisplayName>GlobalVars</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5005</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1005</Reference>
        <Reference ReferenceType="HasModellingRule">ns=0;i=78</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=6007</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=6008</Reference>
  </References>
</UAObject>
```

Listing 5.35: Mapping of the references to the Variables in the Resource - Global Vars XML UAObject

Add the *nGlobal1* and *nGlobal2 Variables* and had the *"HasComponent Reference* pointing to the *CtrlResource GlobalVars Object*.

```
<UAVariable NodeId="ns=3;i=6007" BrowseName="3:nGlobal1" DataType="Boolean">
  <DisplayName>nGlobal1</DisplayName>
  <References>
        <Reference ReferenceType="HasCompoent" IsForward="false">ns=3;i=5008</Reference>
  </References>
</UAVariable>

<UAVariable NodeId="ns=3;i=6008" BrowseName="3:nGlobal2" DataType="Boolean">
  <DisplayName>nGlobal2</DisplayName>
  <References>
        <Reference ReferenceType="HasCompoent" IsForward="false">ns=3;i=5008</Reference>
  </References>
```

```
</UAVariable>
```

Listing 5.36: Mapping of the Resource Global Variables to XML UAVariable

5.3.1.6 Resource - Task and Run-time Program

```
RESOURCE CPU_1 ON CPU_A100
   ...
   TASK task1(INTERVAL := T#5ms,PRIORITY := 0);
   PROGRAM Main1 WITH task1: Main;
 END_RESOURCE
```

Listing 5.37: Example IEC 61131-3 *Resource* based on OPC UA Diagram [17]

To map the *Task* to OPC UA, a *Object* of type *CtrlTask* needs to be created and added has a component of the *Tasks Object* of the *CtrlResource* defined earlier.

```
<UAObject NodeId="ns=3;i=5006" BrowseName="3:Tasks" ParentNodeId="ns=3;i=5005">
  <DisplayName>Tasks</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5005</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1004</Reference>
        <Reference ReferenceType="HasModellingRule">ns=0;i=78</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=5009</Reference>
  </References>
</UAObject>

<UAObject NodeId="ns=3;i=5009" BrowseName="3:task1" ParentNodeId="ns=3;i=5006">
  <DisplayName>task1</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5006</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=2;i=1006</Reference>
    <Reference ReferenceType="HasProperty">ns=3;i=6009</Reference>
    <Reference ReferenceType="HasProperty">ns=3;i=6010</Reference>
  </References>
</UAObject>
```

Listing 5.38: Instantiation of the CtrlTask XML UAObject and addition of References to the Resource - Tasks Object

The *Task* configuration properties - *Priority*, *Simple* and *Interval* - are attached has *Properties* to the *CtrlTask Object*. In the example code, only the *Priority* and *Interval* are specified, so two *Variable* Nodes need to be created and the *Reference* "HasProperty" must be set pointing to the *CtrlTask Object*. This way the *Variables* will be recognized by the server has *Properties* of the *CtrlTask Object*.

```
<UAVariable NodeId="ns=3;i=6009" BrowseName="3:Interval" ParentNodeId="ns=3;i=5009" DataType="String">
  <DisplayName>Interval</DisplayName>
  <References>
        <Reference ReferenceType="HasProperty" IsForward="false">ns=3;i=5009</Reference>
  </References>
</UAVariable>

<UAVariable NodeId="ns=3;i=6010" BrowseName="3:Priority" ParentNodeId="ns=3;i=5009" DataType="UInt32">
  <DisplayName>Priority</DisplayName>
  <References>
        <Reference ReferenceType="HasProperty" IsForward="false">ns=3;i=5009</Reference>
  </References>
  <Value>
        <UInt32>0</UInt32>
  </Value>
</UAVariable>
```

Listing 5.39: Mapping of the CtrlTask Object Properties to the XML UAVariable

Finally, the *Program* associated with the *Task*, the Main *Program*, needs to be instantiated and added has a component of the *CtrlResource Programs Object*. To link the *CtrlProgram* created with the *CtrlTask* defined above, a "With" *Reference* must be set pointing to the *CtrlTask*.

```
<UAObject NodeId="ns=3;i=5007" BrowseName="3:Programs" ParentNodeId="ns=3;i=5005">
  <DisplayName>Programs</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5005</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1004</Reference>
        <Reference ReferenceType="HasModellingRule">ns=0;i=78</Reference>
    <Reference ReferenceType="HasComponent">ns=3;i=5010</Reference>
  </References>
</UAObject>

<UAObject NodeId="ns=3;i=5010" BrowseName="3:Main1" ParentNodeId="ns=3;i=5007">
  <DisplayName>Main1</DisplayName>
  <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=3;i=5007</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=3;i=1002</Reference>
        <Reference ReferenceType="With">ns=3;i=5009</Reference>
  </References>
</UAObject>
```

Listing 5.40: *Program* mapping to XML UAObject

Chapter 6

open62541

open62541 was the chosen OPC UA implementation. [81] It has available documentation and provides tutorials for learning how to implement the most basic functionality.

This project uses the open62541 Server SDK to implement the OPC UA Server for Beremiz Run-time environment, the XML NodeSet Compiler - a tool provided with the open62541 library to translate OPC UA XML into the open62541 C code - and extends the open62541 structures implementing the OPC UA Information Model.

6.1 open62541 OPC UA Information Model

The open62541 stack implements the UA Binary encoding - defined in OPC UA Part 6. The open62541 documentation describes it's C structures implementing the OPC UA Standard-Nodes in https://open62541.org/doc/current/nodestore.html. The following listings describe the open62541 structures that were used to implement the OPC UA Node Classes used during the development phase and are described in 7.3 MatIEC OPC UA Generator - Internal Structure and shown in Figure 7.6.

Some of the UA_NODE_BASEATTRIBUTES are defined in the AbstractNode Class.

```
typedef struct {
    UA_NodeId referenceTypeId;
    UA_Boolean isInverse;
    size_t targetIdsSize;
    UA_ExpandedNodeId *targetIds;
} UA_NodeReferenceKind;

#define UA_NODE_BASEATTRIBUTES \
    UA_NodeId nodeId; \
    UA_NodeClass nodeClass; \
    UA_QualifiedName browseName; \
    UA_LocalizedText displayName; \
    UA_LocalizedText description; \
    UA_UInt32 writeMask; \
    size_t referencesSize; \
    UA_NodeReferenceKind *references; \
                                                \
    /* Members specific to open62541 */ \
    void *context;

typedef struct {
    UA_NODE_BASEATTRIBUTES
} UA_Node;
```

Listing 6.1: open62541 BaseNodeAttributes and UA_Node structure

The UA_ObjectAttributes structure is implemented in the ObjectNode Class.

```
typedef struct {
    UA_UInt32 specifiedAttributes;
    UA_LocalizedText displayName;
    UA_LocalizedText description;
    UA_UInt32 writeMask;
    UA_UInt32 userWriteMask;
    UA_Byte eventNotifier;
} UA_ObjectAttributes;
```

Listing 6.2: open62541 UA_ObjectAttributes structure

The UA_ObjectTypeAttributes structure is implemented in the ObjectTypeNode Class.

```
typedef struct {
    UA_UInt32 specifiedAttributes;
    UA_LocalizedText displayName;
    UA_LocalizedText description;
    UA_UInt32 writeMask;
    UA_UInt32 userWriteMask;
    UA_Boolean isAbstract;
} UA_ObjectTypeAttributes;
```

Listing 6.3: open62541 UA_ObjectTypeAttributes structure

The UA_Variable structure is implemented in the VariabeNode Class.

```
typedef struct {
    UA_UInt32 specifiedAttributes;
    UA_LocalizedText displayName;
    UA_LocalizedText description;
    UA_UInt32 writeMask;
    UA_UInt32 userWriteMask;
    UA_Variant value;
    UA_NodeId dataType;
    UA_Int32 valueRank;
    size_t arrayDimensionsSize;
    UA_UInt32 *arrayDimensions;
    UA_Byte accessLevel;
    UA_Byte userAccessLevel;
    UA_Double minimumSamplingInterval;
    UA_Boolean historizing;
} UA_VariableAttributes;
```

Listing 6.4: open62541 UA_VariableAttributes structure

The UA_VariableType structure is implemented in the VariableTypeNode Class.

```
typedef struct {
    UA_UInt32 specifiedAttributes;
    UA_LocalizedText displayName;
    UA_LocalizedText description;
    UA_UInt32 writeMask;
    UA_UInt32 userWriteMask;
    UA_Variant value;
    UA_NodeId dataType;
    UA_Int32 valueRank;
    size_t arrayDimensionsSize;
    UA_UInt32 *arrayDimensions;
    UA_Boolean isAbstract;
} UA_VariableTypeAttributes;
```

Listing 6.5: open62541 UA_VariableTypeAttributes structure

6.2 OPC UA Server

In Listing 6.6 the code for an empty open62541 OPC UA Server is listed. In the open62541 documentation there is a tutorial on how to build this simple server - https://open62541.org/doc/current/tutorial_server_firsts
In 7.5 OPC UA Server, an example of the server code (with the information models that were utilized in this project added) - used in the project is listed.

```
#include <signal.h>
#include <open62541.h>

UA_Boolean running = true;
static void stopHandler(int sig) {
    UA_LOG_INFO(UA_Log_Stdout,
                UA_LOGCATEGORY_USERLAND,
                "receive ctrl−C");
    running = false;
}

int main(void) {
    // SIGINT: interrupt signal such as ctrl−C
    signal(SIGINT, stopHandler);
    // SIGTERM: termination request
    signal(SIGTERM, stopHandler);

    UA_ServerConfig *config = UA_ServerConfig_new_default();
    UA_Server *server = UA_Server_new(config);

    UA_StatusCode retval;

    // Add the Information Models or Nodes directly before starting the server

    // Start the server
    retval = UA_Server_run(server, &running);

    UA_Server_delete(server);
    UA_ServerConfig_delete(config);

    return (int) retval;
}
```

Listing 6.6: Example open62541 empty server code

6.2.1 Adding Nodes to the OPC UA Server

To add the Nodes to the OPC UA Server, open62541 provides two ways, specified in *Node Addition and Deletion*. One is using the methods for each of the NodeClasses, such as the *UA_Server_addVariableNode*, that are typed versions of the base method *__UA_Server_addNode* that should not be used.

```
UA_StatusCode
__UA_Server_addNode(UA_Server *server,
                    const UA_NodeClass nodeClass,
                    const UA_NodeId *requestedNewNodeId,
                    const UA_NodeId *parentNodeId,
                    const UA_NodeId *referenceTypeId,
                    const UA_QualifiedName browseName,
                    const UA_NodeId *typeDefinition,
                    const UA_NodeAttributes *attr,
                    const UA_DataType *attributeType,
                    void *nodeContext, UA_NodeId *outNewNodeId);

static UA_INLINE UA_StatusCode
UA_Server_addVariableNode(UA_Server *server,
                          const UA_NodeId requestedNewNodeId,
                          const UA_NodeId parentNodeId,
                          const UA_NodeId referenceTypeId,
                          const UA_QualifiedName browseName,
                          const UA_NodeId typeDefinition,
                          const UA_VariableAttributes attr,
                          void *nodeContext, UA_NodeId *outNewNodeId) {
    return __UA_Server_addNode(server,
                               UA_NODECLASS_VARIABLE,
                               &requestedNewNodeId,
                               &parentNodeId,
                               &referenceTypeId,
                               browseName,
                               &typeDefinition,
                               (const UA_NodeAttributes*)&attr,
                               &UA_TYPES[UA_TYPES_VARIABLEATTRIBUTES],
                               nodeContext, outNewNodeId);
}

(* Other methods for each of the remaining NodeClasses *)
```

Listing 6.7: open62541 Node addition method

The other is the pair of methods *UA_Server_addNode_begin* and *UA_Server_addNode_finish*. This is the preferred way when the nodes are modified after being instantiated, such as when references to child nodes instantiated later are added and is used in 6.3 XML NodeSet Compiler.

```
UA_StatusCode
UA_Server_addNode_begin(UA_Server *server, const UA_NodeClass nodeClass,
                        const UA_NodeId requestedNewNodeId,
                        const UA_NodeId parentNodeId,
                        const UA_NodeId referenceTypeId,
                        const UA_QualifiedName browseName,
                        const UA_NodeId typeDefinition,
                        const void *attr, const UA_DataType *attributeType,
                        void *nodeContext, UA_NodeId *outNewNodeId);

UA_StatusCode
UA_Server_addNode_finish(UA_Server *server, const UA_NodeId nodeId);
```

Listing 6.8: open62541 Node addition begin and finish method pair

6.3 XML NodeSet Compiler

The XML NodeSet Compiler is a tool available with the open62541 package and is an important component in this project's design - 7.4 XML NodeSet Compiler. It is written in python and is used to translate OPC UA Information Models in XML, following the OPC UA Nodeset XML Schema - used as a definition to import or export Nodes into a server's *AddressSpace* -, into the open62541 intermediate C code. This intermediate C code - representing the Information Model specified in XML - is then included into the open62541 OPC UA Server code. This code represents a OPC UA Binary Server with the Information Model specified available in it's *AddressSpace*. GUI tools can be used to create Information Models and export them to the UA Nodeset Schema and the XML NodeSet compiler used to convert them into a working server. The XML NodeSet's compilation process for the various information model layers specified in 5 OPC UA Information Model for IEC 61131-3 is described in 7.4 XML NodeSet Compiler. In open62541's online documentation there is a tutorial explaining how to use the XML NodeSet Compiler - https://open62541.org/doc/current/nodeset_compiler.html.

The XML file has a root XML element, the UANodeSet, that represents a *namespace* in the OPC UA Server *AddressSpace*. All the various *Nodes* that form the Information Model are added as child's of the UANodeSet XML element. Listing 6.9 shows the UANodeSet element XML Scheme. Note the 8 Standard Node Classes that this element accepts as a child. The UANodeSet is defined in OPC UA Part 6 [82], Annex F - Information Model XML Schema - and the XSD file containing it is available in the OPC Foundation UA-Nodeset repository [15].

```
<xs:element name="UANodeSet">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="NamespaceUris" type="UriTable" minOccurs="0"></xs:element>
      <xs:element name="ServerUris" type ="UriTable" minOccurs="0"></xs:element>
```

```
        <xs:element name="Models" type="ModelTable" minOccurs="0"></xs:element>
        <xs:element name="Aliases" type ="AliasTable" minOccurs="0"></xs:element>
        <xs:element name="Extensions" type="ListOfExtensions" minOccurs="0"></xs:element>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="UAObject" type="UAObject"></xs:element>
          <xs:element name="UAVariable" type="UAVariable"></xs:element>
          <xs:element name="UAMethod" type="UAMethod"></xs:element>
          <xs:element name="UAView" type="UAView"></xs:element>
          <xs:element name="UAObjectType" type="UAObjectType"></xs:element>
          <xs:element name="UAVariableType" type="UAVariableType"></xs:element>
          <xs:element name="UADataType" type="UADataType"></xs:element>
          <xs:element name="UAReferenceType" type="UAReferenceType"></xs:element>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="LastModified" type="xs:dateTime" use="optional"></xs:attribute>
    </xs:complexType>
  </xs:element>
```

Listing 6.9: UANodeSet XML Schema definition [15]

Listing 6.10 is an example of a XML UANodeSet element representing the Example layer in 5.3 Example XML mapping. Note the various layers required, added in the NamespaceUris and in the RequiredModel of the Model that it's being specified - "192.168.2.65/ControllerServer". The Nodes specified in 5.3 Example XML mapping are added bellow the Aliases element. The Aliases element is used to add string aliases representing certain nodes and it's mainly used to set string aliases for certain *ReferenceTypes*. This way, when defining a reference, in the *ReferenceType*, a string can be used instead of the NodeId.

The order used in the NamespaceUris follows the order specified in section 7.3 of the OPC UA Information Model for IEC 61131-3. Because the *Nodes* specified in the example mapping only use the upper layer - OPC UA IEC 61131-3, to extend and implement the *ObjectTypes* and use the *References* defined - and the layer above that - OPC UA DI, to add the *CtrlConfiguration* as a *Component* of the *DeviceSet* - the Namespace for the basic layer could not need to be specified, and the Examples could be added in the same *namespace* as the OPC UA IEC 61131-3, but OPC Foundation recommends to use a modular layer approach and add the *real-world* implementation of the OPC UA IEC 61131-3 as a new layer, called *ControllerServer*, as was done in the 5.3 Example XML mapping - Figure 5.1. Optionally, you could create a new namespace for every *CtrlResource* or *CtrlFunctionBlock* implemented in the *ControllerServer*.

```
<UANodeSet xmlns="http://opcfoundation.org/UA/2011/03/UANodeset.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <NamespaceUris>
        <Uri>http://opcfoundation.org/UA</Uri>
        <Uri>http://192.168.2.65/ControllerServer</Uri>
        <Uri>http://opcfoundation.org/UA/DI/</Uri>
        <Uri>http://PLCopen.org/OpcUa/IEC61131-3/</Uri>
```

```
    </NamespaceUris>
    <Models>
        <Model ModelUri="http://192.168.2.5/ControllerServer"
               PublicationDate="2010-03-24T00:00:00Z"
               Version="1.00">
            <RequiredModel ModelUri="http://PLCopen.org/OpcUa/IEC61131-3/"
                           PublicationDate="2010-03-24T00:00:00Z"
                           Version="1.00"/>
            <RequiredModel ModelUri="http://opcfoundation.org/UA/"
                           PublicationDate="2018-02-09T00:00:00Z"
                           Version="1.04"/>
            <RequiredModel ModelUri="http://opcfoundation.org/UA/DI/"
                           PublicationDate="2012-12-31T00:00:00Z"
                           Version="1.01"/>
        </Model>
    </Models>
    <Aliases>
        <Alias Alias="Boolean">ns=0;id=1</Alias>
        <Alias Alias="Byte">ns=0;id=3</Alias>
        <Alias Alias="ByteString">ns=0;id=15</Alias>

        <!-- More Alias -->

        <Alias Alias="UInteger">ns=0;id=28</Alias>
        <Alias Alias="XmlElement">ns=0;id=16</Alias>
    </Aliases>

    <!-- Add the OPC UA XML Nodes -->

</UANodeSet>
```

Listing 6.10: UANodeSet XML Example

The output of the XML NodeSet Compiler is a C source and header file. The C source has a main method, used to add all the Nodes to the OPC UA Server. The header file can be included in the OPC UA Server code and the main method added before starting the server as described in 7.5 OPC UA Server.

Each Node is mapped into to two methods, the first method, the "begin" one, and the second method, the "finish" one - described in Listing 6.8. This is required because of the *component* Nodes attached to the parent Node that are added after it. The references connecting the parent Node with it's component Nodes are added in the component Node, using the UA_Server_addReference. This methods add the Node to the OPC UA Server in the specified Information Model layer, or *namespace*, as the method takes as arguments the pointers to the UA_Server object and the UA_UInt16 *namespace*. The number specified for each of the two methods represents the order of that Node in the XML NodeSet.

```
<UAObjectType NodeId="ns=1;i=1001" BrowseName="1:ExampleObjectType" IsAbstract="false">
    <DisplayName>ExampleObjectType</DisplayName>
```

```
    <Description>ExampleObjectType</Description>
    <References>
      <Reference ReferenceType="HasComponent">ns=1;i=5002</Reference>
      <Reference ReferenceType="HasSubtype" IsForward="false">i=58</Reference>
    </References>
</UAObjectType>

<UAObject NodeId="ns=1;i=5002" BrowseName="1:ExampleObject" ParentNodeId="ns=1;i=1001">
    <DisplayName>ExampleObject</DisplayName>
    <Description>ExampleObject</Description>
    <References>
      <Reference ReferenceType="HasTypeDefinition">i=58</Reference>
      <Reference ReferenceType="HasModellingRule">i=80</Reference>
      <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=1001</Reference>
    </References>
</UAObject>
```

Listing 6.11: *ExampleObjectType* Node and respective *Component* Node *ExampleObject* in XML

If the first two Nodes in our Listing 6.10 are the *ExampleObjectType* and it's respective child Node *ExampleObject*, as described in Listing 6.11, in the source C file generated by the XML NodeSet Compiler, two functions are generated for each Node. In the first function, the "begin" one, the Node attributes are instantiated, using the Attributes structure for the respective NodeClass, and are added to the OPC UA Server using the method *UA_Server_addNode_begin*. The second function, the "finish" one, finishes the addition of the Node to the server, using the *UA_Server_addNode_finish*.

```
/* ExampleObjectType - ns=1;i=1001 */

static UA_StatusCode function_ua_namespace_example_0_begin(UA_Server *server,
                                                           UA_UInt16* ns) {
  UA_StatusCode retVal = UA_STATUSCODE_GOOD;
  UA_ObjectTypeAttributes attr = UA_ObjectTypeAttributes_default;
  attr.isAbstract = true;
  attr.displayName = UA_LOCALIZEDTEXT("", "ExampleObjectType");
  attr.description = UA_LOCALIZEDTEXT("", "ExampleObjectType");
  attr.writeMask = 0;
  attr.userWriteMask = 0;
  retVal |= UA_Server_addNode_begin(server, UA_NODECLASS_OBJECTTYPE,
  UA_NODEID_NUMERIC(ns[1], 1001),
  UA_NODEID_NUMERIC(ns[0], 58),
  UA_NODEID_NUMERIC(ns[0], 45),
  UA_QUALIFIEDNAME(ns[1], "ExampleObjectType"),
  UA_NODEID_NULL,
```

```
    (const UA_NodeAttributes*)&attr,
    &UA_TYPES[UA_TYPES_OBJECTTYPEATTRIBUTES],
    NULL, NULL);
    return retVal;
}

static UA_StatusCode function_ua_namespace_example_0_finish(UA_Server *server,
                                                            UA_UInt16* ns) {
    return UA_Server_addNode_finish(server,
        UA_NODEID_NUMERIC(ns[1], 1001)
    );
}

/* ExampleObject - ns=1;i=5002 */

static UA_StatusCode function_ua_namespace_example_1_begin(UA_Server *server,
                                                           UA_UInt16* ns) {
    UA_StatusCode retVal = UA_STATUSCODE_GOOD;
    UA_ObjectAttributes attr = UA_ObjectAttributes_default;
    attr.displayName = UA_LOCALIZEDTEXT("", "ExampleObject");
    attr.description = UA_LOCALIZEDTEXT("", "ExampleObject");
    attr.writeMask = 0;
    attr.userWriteMask = 0;
    retVal |= UA_Server_addNode_begin(server, UA_NODECLASS_OBJECT,
    UA_NODEID_NUMERIC(ns[1], 5002),
    UA_NODEID_NUMERIC(ns[1], 1001),
    UA_NODEID_NUMERIC(ns[0], 47),
    UA_QUALIFIEDNAME(ns[1], "ExampleObject"),
    UA_NODEID_NUMERIC(ns[0], 58),
    (const UA_NodeAttributes*)&attr,
    &UA_TYPES[UA_TYPES_OBJECTATTRIBUTES],
    NULL, NULL);
    // Add HasModellingRule Reference
    retVal |= UA_Server_addReference(server,
                                     UA_NODEID_NUMERIC(ns[1], 5002),
                                     UA_NODEID_NUMERIC(ns[0], 37),
                                     UA_EXPANDEDNODEID_NUMERIC(ns[0], 80), true);
    return retVal;
}

static UA_StatusCode function_ua_namespace_example_1_finish(UA_Server *server,
                                                            UA_UInt16* ns) {
    return UA_Server_addNode_finish(server,
    UA_NODEID_NUMERIC(ns[1], 5002)
```

```
  );
}
```

Listing 6.12: *ExampleobjectType* Node and respective *Component* Node *ExampleObject* in open62541 C code representation created by the XML NodeSet Compiler

The generated source file main method is used to include the *namespace* in the server. For each Node declared in the source file, the "begin" function is called by ascending order. The "finish" function is called by descending order so that the references to parent Nodes, added first, are set correctly in the server.

```
UA_StatucCode ua_namespace_example(UA_Server * server) {
    UA_UInt16[4];
    ns[0]= UA_Server_addNamespace(server, "http://opcfoundation.org/UA/");
    ns[1]= UA_Server_addNamespace(server, "http://192.168.2.65/ControllerServer/");
    ns[2]= UA_Server_addNamespace(server, "http://opcfoundation.org/UA/DI/");
    ns[3]= UA_Server_addNamespace(server, "http://PLCopen.org/OpcUa/IEC61131-3/");

    retVal |= function_ua_namespace_example_0_begin(server, ns);
    retVal |= function_ua_namespace_example_1_begin(server, ns);

    (...)

    retVal |= function_ua_namespace_example_1_finish(server, ns);
    retVal |= function_ua_namespace_example_0_finish(server, ns);
    return retVal;
}
```

Listing 6.13: Main method generated by the XML NodeSet Compiler

Chapter 7

Design

In this chapter the Design approach to the objective of providing OPC UA support to Beremiz is described. After that the software developed and how the preexisting software tools were used are detailed. The work developed was only possible because of the specification described in 5 OPC UA Information Model for IEC 61131-3. The software tools of the open62541 OPC UA implementation - described in chapter 6 open62541 - used in the development of this work were previously described in sections 6.2 OPC UA Server and 6.3 XML NodeSet Compiler.

7.1 First Architecture

To provide Beremiz with OPC UA support the Beremiz Compilation Process Architecture, represented in Figure 4.2 was upgraded with an OPC UA Server. For the mapping of the IEC 61131-3 program to the OPC UA Information Model the MatIEC compiler was upgraded and the XML NodeSet Compiler, described in 6.3 XML NodeSet Compiler, was used as an intermediary. The new Compilation Process Architecture is represented in Figure 7.2 was designed.

The MatIEC Compiler was upgraded with a UA XML code generation phase module. The MatIEC receives the IEC 61131-3 text format code as input and the new module maps the IEC 61131-3 elements into OPC UA Nodes following the OPC UA Information Model for IEC 61131-3 - described in 5 OPC UA Information Model for IEC 61131-3 - and parses them into a xml file following the OPC UA Information Model XML Schema - OPC UA Part 6, section 5.3. The generated xml file represents a layer of the Information Model - a *namespace* of the OPC UA Server. Because the OPC UA Information Model XML Schema is a normalized format defined by the OPC Foundation in the OPC UA specification, it provides a compatibility format. It can be used with Graphical Modelling Tools for OPC UA, is a human-friendly format, and enables the *namespace* to be imported into different OPC UA Servers.

The next component, the open62541 XML NodeSet Compiler, receives the xml file and translates it's Information Model into the open62541 intermediate C code representation. The C code that the NodeSet Compiler's outputs is included in the open62541 OPC UA Server code. This

code is compiled alongside the softPLC C code into machine-code by the gcc compiler. The generated binary - *.out - when executed, starts the softPLC, alongside the OPC UA Server with the mapped Information Model.

7.1.1 MatIEC OPC UA XML Code Generator

Since the OPC UA Information Model for IEC 61131-3 specification describes how to correctly map all the main IEC 61131-3 model elements into the OPC UA Information model, and that the MatPLC IEC Compiler - described in 4.2 MatIEC Compiler - translates the IEC 61131-3 textual languages to C code, it was possible to upgrade MatIEC to also translate this languages to an OPC UA format. Because of the modular architecture of the MatIEC, it was only needed to had a new Code Generation phase module - besides the ones described in 4.2.3 MatIEC. And the phases of the front-end of the compiler didn't need to be upgraded. MatIEC structure, with the addition of this new module, is shown in Figure 7.1

The UA XML format, as stated earlier, was chosen with the objective of using the open62541 XML NodeSet Compiler and providing a compatibility format to other tools. This Code Generation module, maps the IEC 61131-3 elements described as symbols in the MatIEC Abstract Syntax Tree - 4.2 MatIEC Compiler - to OPC UA Nodes according to the OPC UA Information Model for IEC 61131-3 specification - 5 OPC UA Information Model for IEC 61131-3 - and adds them to a in-memory structure of OPC UA Nodes. After all the elements have been mapped to OPC UA Nodes and added to the Nodes structure a serialization phase iterates the Nodes structure, serializes the Nodes to XML and generates the XML file.

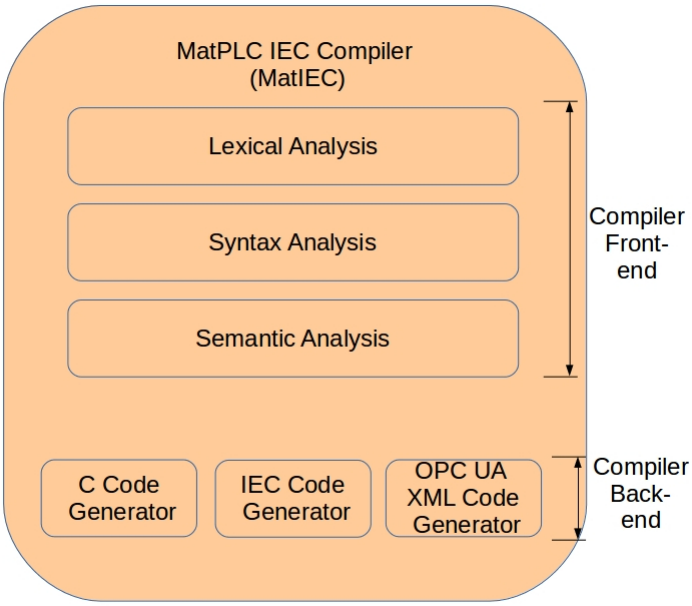

Figure 7.1: MatIEC Compiler Structure with the new OPC UA XML Code Generation

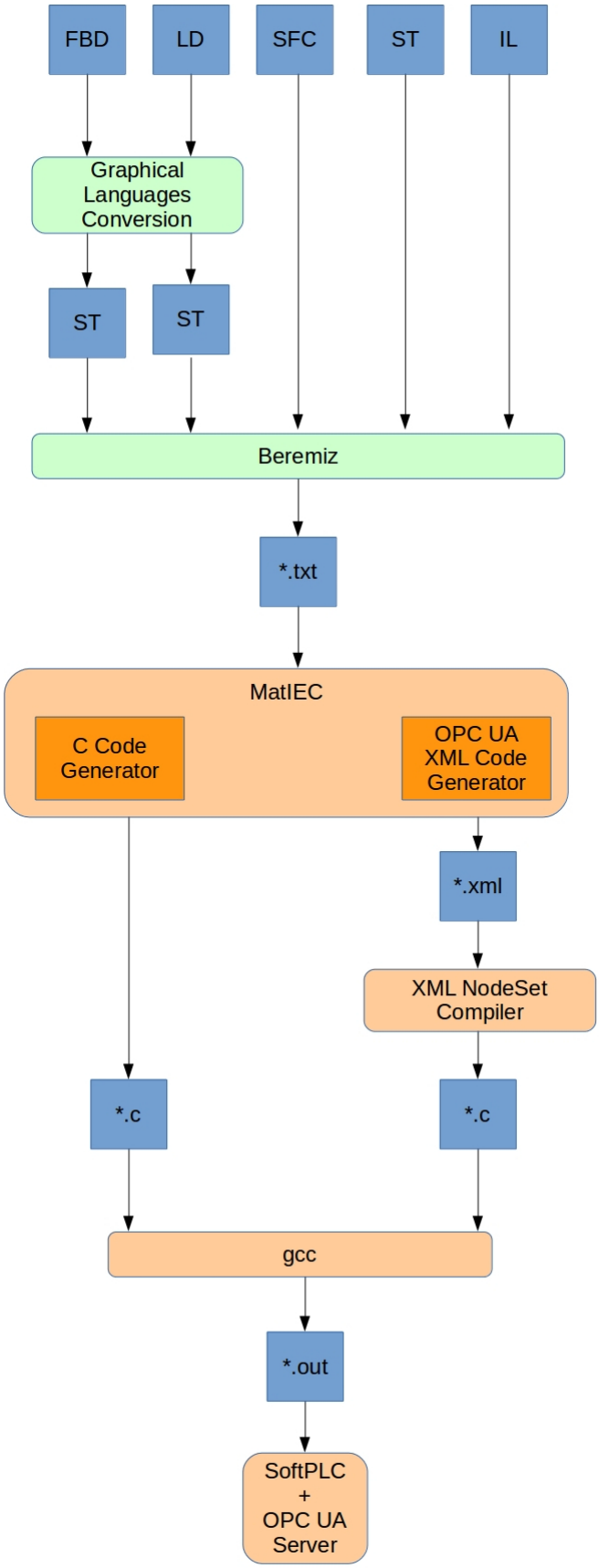

Figure 7.2: First Architecture

7.2 Second Architecture

The previous Architecture relied on the XML NodeSet Compiler as a middle-man between the MatIEC and the OPC UA Server, because the MatIEC generated the translation of the IEC 61131-3 program to OPC UA using the OPC UA XML format. This middle step increased the complexity of the compilation process and it was difficult to automate the task of translating the XML code using the XML NodeSet Compiler and including the translated code into the OPC UA Server. If the MatIEC could translate the program directly to the open62541 OPC UA C representation it was possible to remove this middle-step. This way the compilation process could be done automatically. The MatIEC could map the program directly to the open62541 C representation of the UA namespace and could generate the entire OPC UA Server with all the namespaces needed. This second Architecture, without the XML NodeSet Compiler intermediate step, is shown in Figure 7.4.

7.2.1 MatIEC OPC UA open62541 C Generator

This new code generation module is basically the previous Code Generation Phase used for the UA XML generation but with a different serialization format. After the IEC 61131-3 elements have been mapped into OPC UA Nodes and added to the Nodes memory structure, instead of serializing the Nodes into XML the Nodes are serialized into open62541 C code.

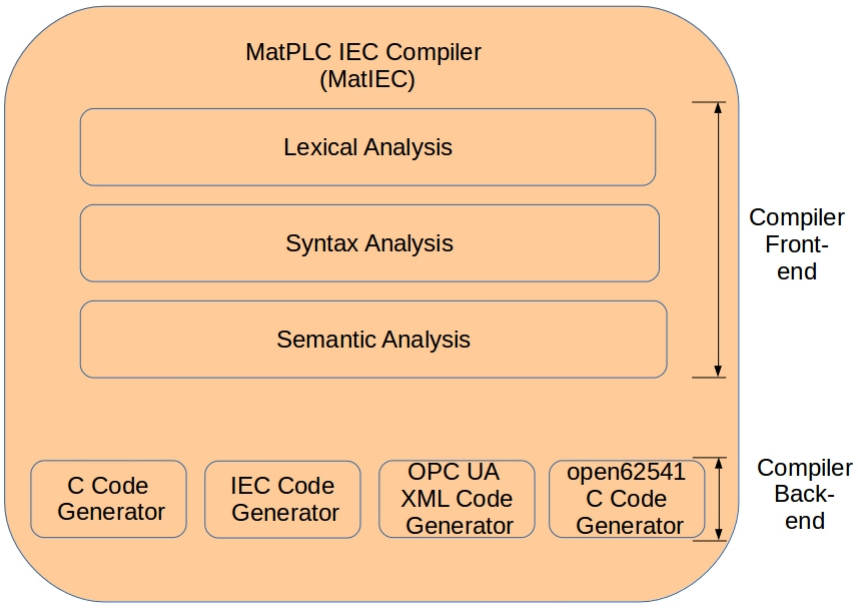

Figure 7.3: MatIEC Compiler Structure with the new open62541 C Code Generator

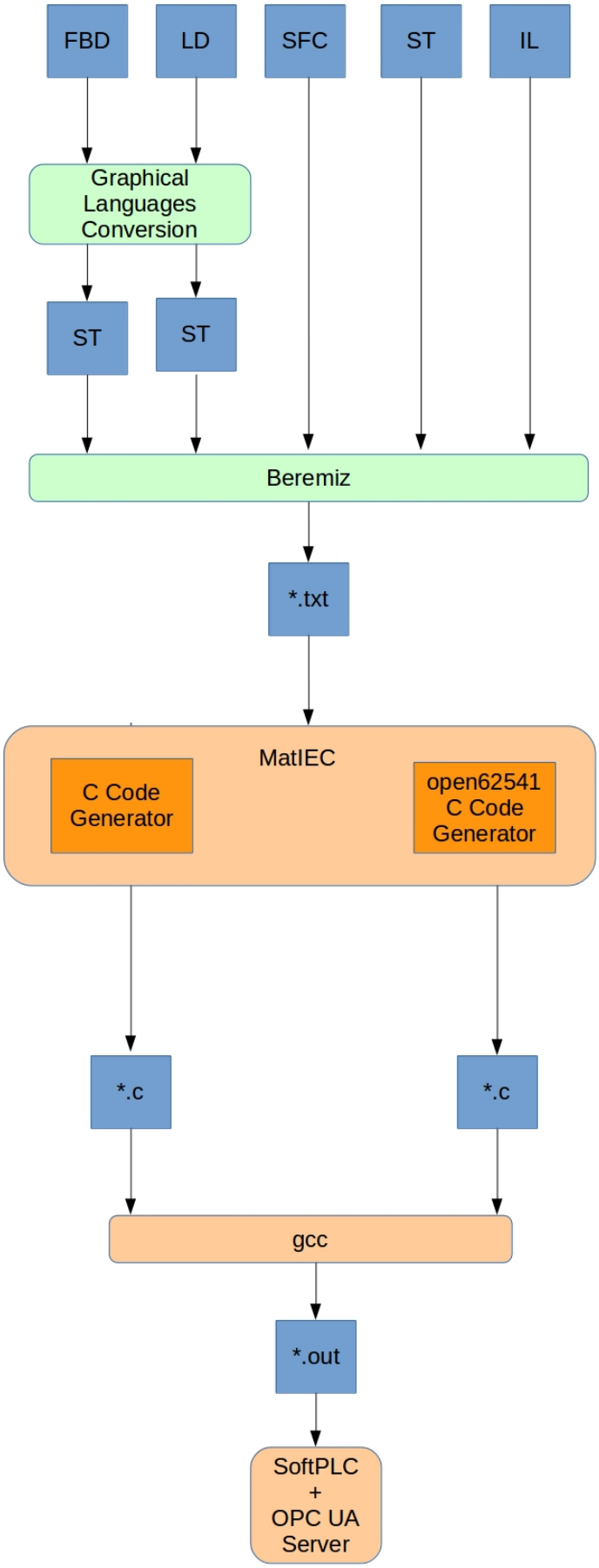

Figure 7.4: Second Architecture

7.3 MatIEC OPC UA Generator - Internal Structure

During the Analysis part of the MatIEC compiler - it's front-end - the source IEC 61131-3 program is broken down into symbols that are organized into an object structure, the Abstract Syntax Tree - 4.2.3 MatIEC. This Abstract Syntax Tree was designed using the Visitor Pattern. This pattern enables a decoupling between the object structure and the operations to be performed on it. The various classes composing the Abstract Syntax Tree structure have a method that accepts the Visitor. The Visitor is defined as an Interface with methods to Visit the various classes that implemented the accepting method. This way, various concrete Visitors can be defined, by implementing the Visitor interface, to perform different types of operations without the need to modify the Abstract Syntax Tree. The MatIEC Abstract Syntax Tree is defined inside the absyntax/ directory in the absyntax.hh file. The Visitor interface is defined in the visitor.hh file also inside the absyntax/ directory.

In the Code Generation phase a concrete Visitor is defined, by implementing the Visitor interface, to read the symbols of the Abstract Syntax Tree and print them out within the target program being generated. In this Code Generator in particular, before printing out the target program, OPC UA Nodes need to be created, and the symbols of the Abstract Syntax Tree are used as some of the Attributes of this Nodes.

The Visitor implements the mapping logic according to the OPC UA Informatin Model for IEC 61131-3 to map the IEC 61131-3 elements into it's respective set of OPC UA Nodes. This Nodes have to be added in order according to the other Nodes that they reference. Because of this cross-reference between Nodes, the MatIEC module needs to keep a memory structure of Nodes (the OPC UA NodeStore), where the Nodes can be added with a certain order and later modified to add attributes or references.

During the Visitor life-cycle this loop of visiting the Abstract Syntax Tree, building the Nodes and adding them to the OPC UA NodeStore occurs until all the accepted symbols implemented in the visitor are mapped to OPC UA Nodes. In the end of the Visitor life-cycle, when all the nodes have been mapped and added to the Nodes structure, this structure is iterated and each Node is serialized according to the OPC UA Generator called. If the OPC UA XML Generator is used, the serialization phase will serialize the Nodes into XML, and if the OPC UA open62541 C Generator is used, the serialization phase serializes the Nodes into the open62541 C representation. This process is described in Figure 7.5.

Following on the OPC UA Data Model, an Abstract Class, representing a Node, as in the open62541 UA_Node structure was created. This class is never instantiated and is used as an Interface to be extended by concrete Nodes and is the Interface used in the OPC UA NodeStore. This class was named AbstractNode class. It is implemented by concrete classes representing concrete Nodes that can be instantiated, just like in the OPC UA Data Model. This class implements the variables needed for the Node addition, described in 6.2.1 Adding Nodes to the OPC UA Server, because this is the Interface class used in the in-memory structure where all the Nodes are added.

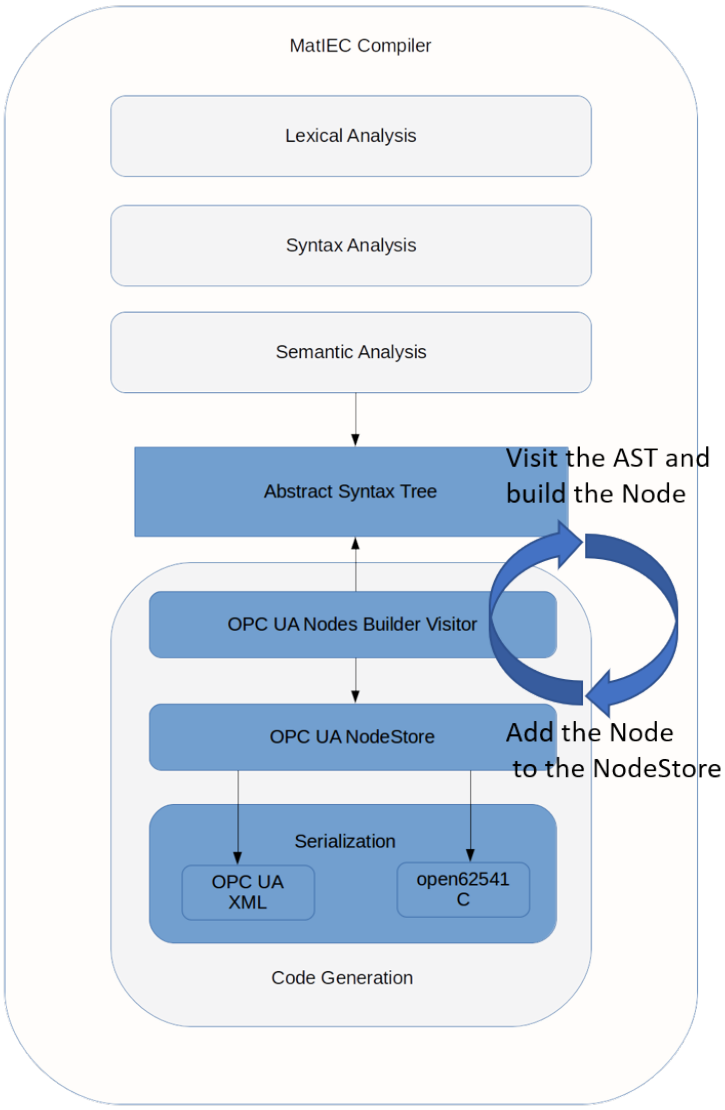

Figure 7.5: MatIEC OPC UA Generator Internal Structure

The other attributes common to all Nodes - such as the *DisplayName*, *Description*, *WriteMask* and *UserWriteMask* - are not implemented in the AbstractNode. For each of concrete classes that extend the AbstractNode class the open62541 Attributes structures - described in 6.1 open62541 OPC UA Information Model - specific for each NodeClass already implement this attributes alongside the ones specific for each different NodeClass.

For example, the ObjectNode is a class that extends the AbstractNode Class, to represent the OPC UA Object Node. By extending the AbstractNode class the ObjectNode class automatically implements it's parent AbstractNode variables needed for the addition to the OPC UA Server method. The ObjectNode has a new attribute, the UA_ObjectAttributes open62541 structure - Listing 6.2 - that contains all the attributes specific of the Object NodeClass and the *DisplayName*, *Description*, *WriteMask* and *UserWriteMask*. The ObjectTypeNode follows the same principle and has a new attributes variable the open62541 UA_ObjectTypeAttributes - Listing 6.3 -, the VariableNode implements the UA_VariableAttributes - Listing 6.4 - and the VariableTypeNode implements the UA_VariableTypeAttributes - Listing 6.5. An UML Class Diagram of the AbstractNode Class and it's child concrete Node Classes is shown in Figure 7.6.

The UA_Reference is a class used to represent OPC UA References. As in the OPC UA Data Model this class as a sourceId variable, representing the NodeId of the source Node, a refTypeId variable, representing the NodeId of the Reference Type of this reference, a targetId representing the NodeId of the target Node and a isForward variable, a boolean flag to specify the direction of the reference. The AbstractNode has a vector of UA_Reference to represent it's associated references to other Nodes. This way helper methods could be created to easily add References between Nodes and during the serialization phase the serialization methods can access this vector to add the References in the serialized Node.

The AbstractNode has the virtual methods for serialization, that are overridden in each of the concrete Node Classes. This is used so that during the serialization phase, the in-memory structure containing pointers to the parent AbstractNode Class of the concrete Nodes, can be iterated and the serialization method called. The method is re-defined to take into account the specific attributes of each NodeClass. The method used in the OPC UA XML Generator to serialize the Nodes is the addNode2XMLElement(). This method uses the tinyxml2 parser. The XMLDocument is an object defined by tinyxml2 and represents the xml file that is gonna be generated. The XMLElement is a pointer to the UANodeSet XML element where the Nodes are added. The UANodeSet is defined in OPC UA: Part 6 and a description of this element is given on the 6.3 XML NodeSet Compiler. The Alias is a pointer to a map<string,UA_NodeId> used to generate the Alias element of the UA_NodeSet and also used in an helper method to find NodeIds kept in this map, by providing the alias name as a parameter. This three parameters that the XML serialization method takes as input are kept by the helper class UA_XML_NodeSet, shown in Figure 7.7.

The methods used in the OPC UA open62541 C Generator to serialize the Nodes are the addNode2UA_Server_begin_print() and the addNode2UA_Server_finish_print(). This two methods print the same result as the two methods printed by the XMLNodeSet Compiler, described in Listing 6.12. The str pointer to a String is the parameter containing the source file String were

the method appends the print version of the addition. The name parameter and index parameter are used to declare the function name. The method addNode2UA_Server_print() is just a helper method that takes the exact same parameters and calls the begin_print and the finish_print method. The helper Class responsible for the serialization, that keeps the str parameter that is passed as a pointer to this serialization methods is the UA_open62541_Namespace shown in Figure 7.7.

The clone() method is used to copy the Node. It is a virtual method re-implemented in each of the concrete classes. This is used when the Node object instantiated is re-used in the Visitor. A clone of the Node is added to the Nodes structure and the Node object can be re-used.

In Figure 7.7 the utility classes used are shown. The NodeId_Counter class is used to keep the a counter of the NodeId for each of the concrete Node Classes and has methods to get the NodeId. This methods are passed as a parameter during the creation of the Node. The UA_Controller class is a helper class to keep the namespaces used.

The UA_NodeStore is the class representing the memory structure of Nodes where the Nodes are added after being created. It is essentially a vector of pointers to the AbstractNode class. It has a method for addition of the Node to the structure and some methods for retrieving it. To add a concrete class Node, for example an ObjectNode, the reference to it's pointer class is added. This enables the vector structure to keep all the Node concrete classes. Because the UA_NodeStore only stores the pointer to the Node it is possible to modify the Node after it was added. If the Node needs to be re-used and has to be added only when it is completely created, the clone method can be used has it was stated above.

The UA_XML_NodeSet is the class responsible for handling the generation of the XML file. It has the tinyxml2 XMLDocument object as a variable, used to generate the XML file, the XMLElement UANodeSet. As it was stated above, this two parameters are passed to the serialization method during the serialization phase alongside the Alias parameter. The Alias and the other variables represent the elements used to define the UANodeSet as described in Listing 6.10. The method init_nodeset() is called in the beginning of the Visitor life-cycle to initialize the UANodeSet. In the end of the Visitor life-cycle the Nodes are serialized to XML and added to the UANodeSet and after that the generate_xml() method is called to generate the xml file.

The UA_open62541_Namespace is the class responsible for handling the generation of the header and source open62541 C files. The source_str String variable is the String parameter passed as a pointer to the open62541 serialization methods were the C code for each Node is added. In the beginning of the the Visitor life-cycle the stream variables are initialized using the open() method, creating an empty source and an empty header file. In the end of the Visitor life-cycle the Nodes are serialized and the C code for the header file is appended to the header_str String variable. After that, the source_str String variable containing the complete source file code and the header_str String variable containing the complete header file code, are bind to it's respective streams using the bind_str() method. The close() method is used to close the streams.

In Figure 7.8 some helper classes used by the Visitor are described. This classes are analogous

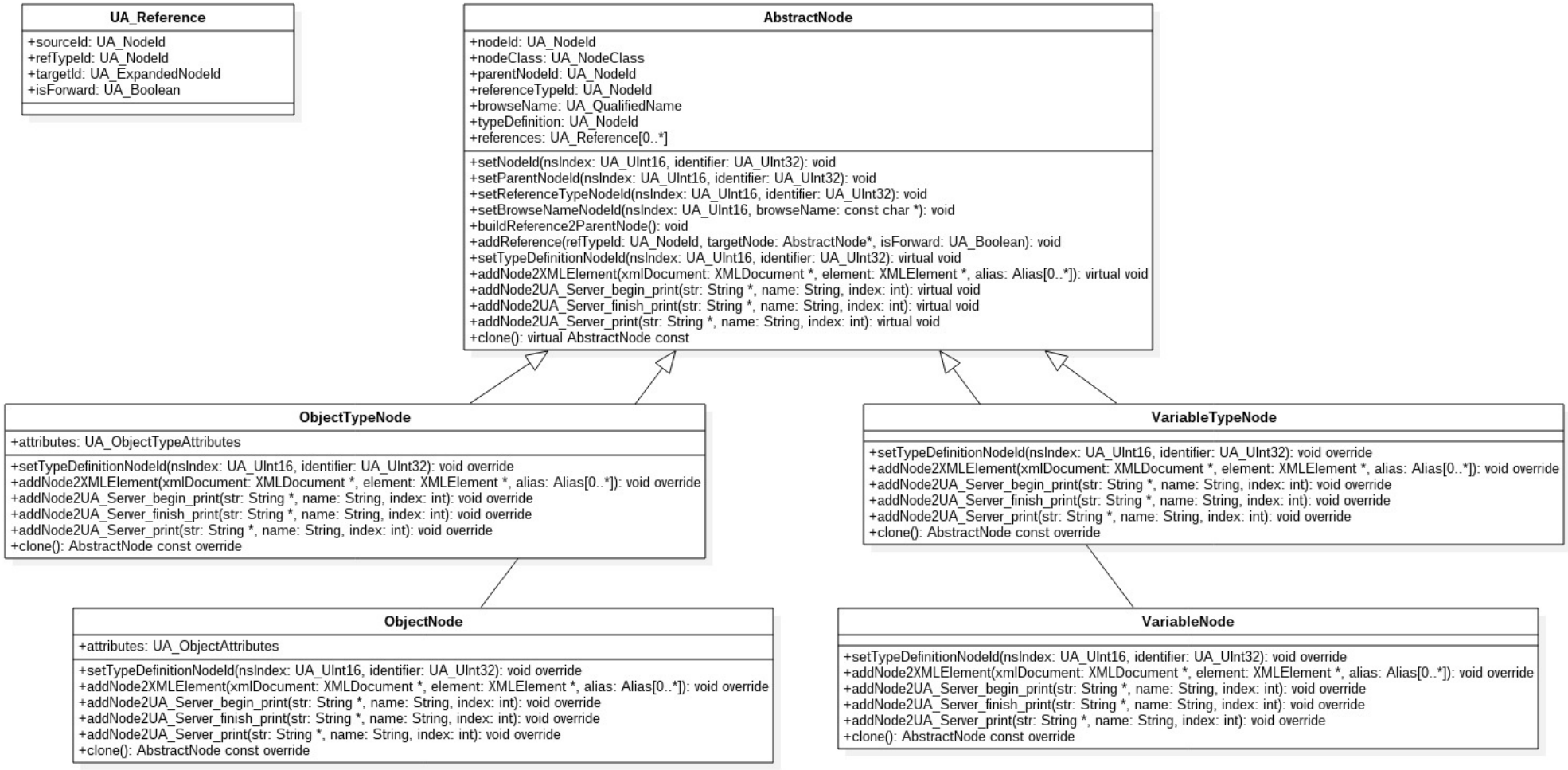

Figure 7.6: UML Class Diagram - OPC UA Nodes

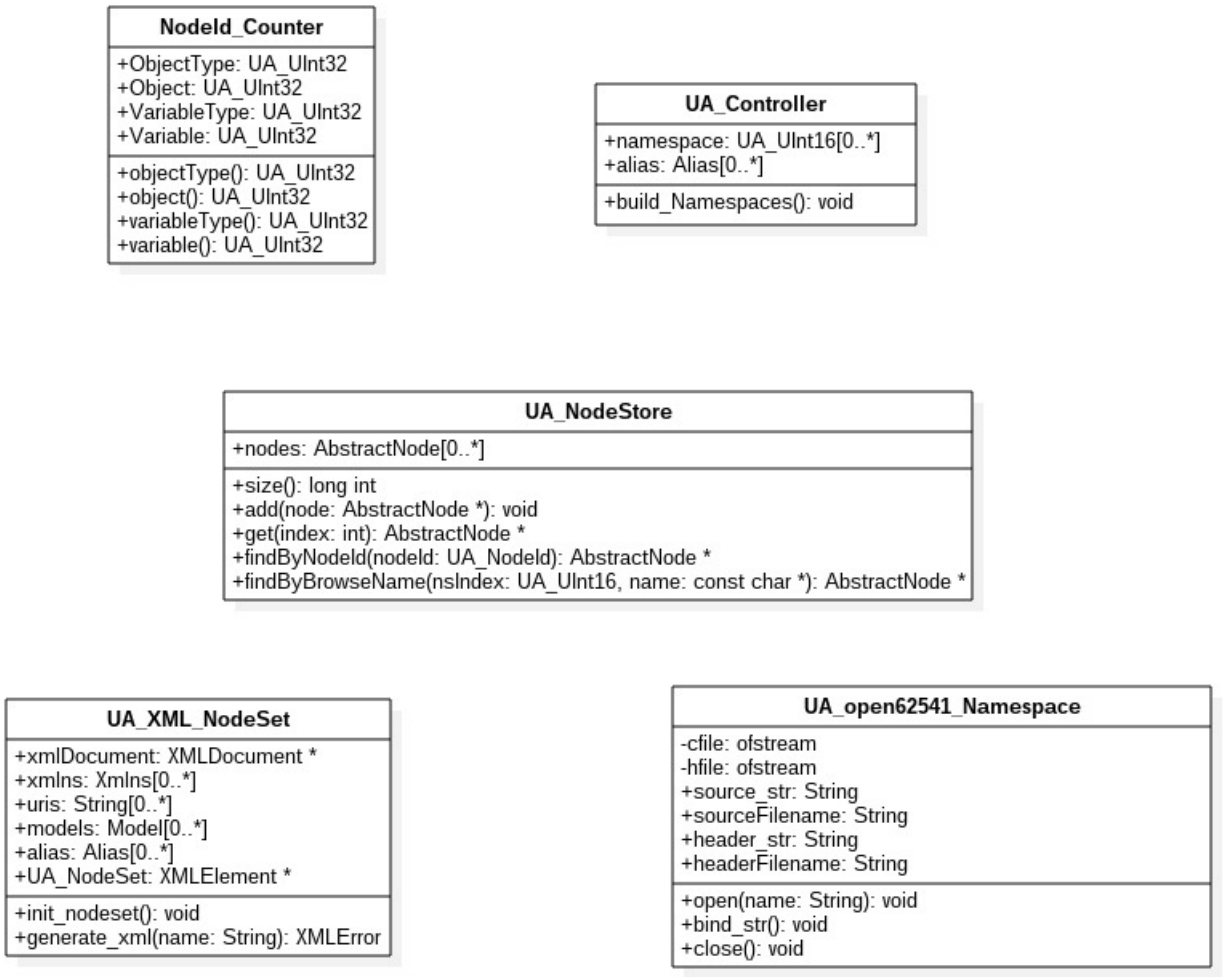

Figure 7.7: UML Class Diagram - Utility Classes

to the set of OPC UA Nodes used to define the IEC 61131-3 elements in the OPC UA Data Model. In Chapter 5 OPC UA Information Model for IEC 61131-3 the ObjectTypes Nodes and component Object Nodes this classes are based on, are defined. This helper classes are composed of an ObjectType and is component Object Nodes, and have helper methods to build the various Nodes. This helper classes were defined with the objective of keeping the least amount of logic possible inside the Visitor. Their defined as variables of the Visitor implemented, enabling them to be called by any of the Visitor visiting methods. This classes are re-used, so their reconstructed every time their called and clones of their Nodes added to the Nodes memory structure.

All this classes have a variable containing a pointer to the UA_Controller Class and another containing a pointer to the UA_NodeStore Class, that are passed down in the constructor of the class. The pointer to the UA_Controller Class is used to pass the namespaces stored in the UA_Controller during the creation of the nodes, when the build methods are called. The pointer to the UA_NodeStore Class is used to add all the Nodes defined in each class to the Nodes memory structure, by calling the add2NodeStore() method.

Because the CtrlResources defined need to be added to it's respective CtrlConfiguration Resources ObjectNode has components, the UA_CtrlResource has a variable containing a pointer to the UA_CtrlConfiguration class. This pointer to the UA_CtrlConfiguration is passed down in the constructor of the UA_CtrlResource. This way when the UA_CtrlResource ObjectTypeNode is created the "HasComponent" Reference to the UA_CtrlConfiguration Resources ObjectNode is automatically added both in the Resources Node and in the CtrlResourceType Node. The same principle is followed in the UA_CtrlTask class, because it's ObjectTypeNode must be added has a component of the UA_CtrlResource Tasks ObjectNode.

The UA_CtrlProgram is used to build CtrlProgram ObjectTypeNodes and it's respective component Nodes. Variable Nodes that are part of a CtrlProgram ObjectTypeNode are not defined in this class because their are built in different visiting method. Variable Nodes can be added by using the addReference method. CtrlPrograms appear before the Configuration Elements - CtrlConfiguration, CtrlResource and CtrlTask - during the Visitor life-cycle. When the CtrlTask is being built using the UA_CtrlTask Class, to add the respective Run-time program associated with it, that was previously built and added to the nodes structure, the findByBrowseName method of the UA_NodeStore is used to retrieve the NodeId of the CtrlProgram. With the NodeId of the CtrlProgram the addReference method is used to add the "With" Reference to the UA_CtrlTask ObjectTypeNode and to add the "HasComponent" Reference to the UA_CtrlResource Programs ObjectNode.

The CtrlFunctionBlock appears before the CtrlProgram and Configuration elements. It's ObjectType Node is built by the UA_CtrlFunctionBlock helper class. The VariableNodes are built in a different visiting stage and added to the UA_CtrlFunctionBlock ObjectTypeNode using the addReference method. Both in the UA_CtrlProgram and in the UA_CtrlFunctionBlock, when the variable being built corresponds to another CtrlFunctionBlock, the NodeId of this element is retrieved from the Nodes structure and the respective reference built.

Figure 7.8: UML Class Diagram - OPC UA IEC 61131-3 Nodes Builder Utility Classes

To use the MatIEC OPC UA Generator with the XML serialization, call:

```
./iec2opc_ua -O -xml="target_name" <input_file>
```

And a <target_name>_nodeset.xml file will be generated.

To use the MatIEC OPC UA Generator with the open62541 C serialization, call:

```
./iec2opc_ua -O -open62541="target_name" <input_file>
```

And the ua_<target_name>_namespace.c and ua_<target_name>_namespace.h files will be generated.

7.4 XML NodeSet Compiler

The XML NodeSet Compiler - described in 6.3 XML NodeSet Compiler - is used to translate the XML Information Models into open62541 C code. From the Information Model layers described in 5.1 Specifications, only the OPC UA XML Information Model, the first layer, doesn't need to be compiled because the open62541 already implements it, all the other three need to be compiled. For each Information Model being compiled, it's parent Information Models need to be linked in the process. Let's go through the process of generating the sources for each of the three Information Models we need.

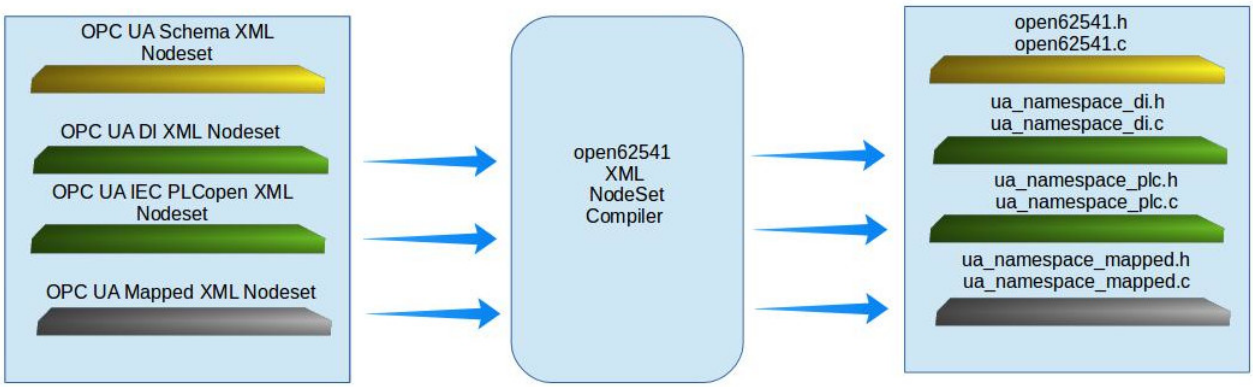

Figure 7.9: open62541 XML NodeSet Compiler

The XML NodeSet Compiler is available with the open62541 implementation. Start by cloning the open62541 repository.

```
git clone https://github.com/open62541/open62541.git
cd open62541−master/
```

The DI XML Information Model, the second layer, as associated data types. First we compile this data types, specified in the OpcUaDiModel.csv and in the Opc.Ua.Di.Types.bsd using the generate_datatypes python script.

```
python tools/generate_datatypes.py
−−namespace=2
−−type−csv=deps/ua−nodeset/DI/OpcUaDiModel.csv
−−type−bsd=deps/ua−nodeset/DI/Opc.Ua.Di.Types.bsd
−−no−builtin
../opcua−open62541−server/src/nodeset/ua_types_di
```

After that we compile the OPC UA DI XML Information Model. This layer extends the OPC UA Information Model layer and uses the specified data types, so they need to be linked in the process. This is done by using the "–existing" flag.

```
python tools/nodeset_compiler/nodeset_compiler.py
−−internal−headers
−−types−array=UA_TYPES
−−types−array=UA_TYPES_DI
−−existing deps/ua−nodeset/Schema/Opc.Ua.NodeSet2.xml
−−xml deps/ua−nodeset/DI/Opc.Ua.Di.NodeSet2.xml
../opcua−open62541−server/src/nodeset/ua_namespace_di
```

After compiling the OPC UA DI types and information model we end up with a ua_types_di.h file alongside the ua_namespace_di.h and ua_namespace_di.c. To generate the IEC 61131-3 Information Model, it needs to be linked with the two Information Models it extends - the OPC UA and the OPC UA DI Information Models. The output will be a ua_namespace_plc.h and ua_namespace_plc.c file representing the OPC UA IEC 61131-3 Information Model.

```
python tools/nodeset_compiler/nodeset_compiler.py
−−internal−headers
−−types−array=UA_TYPES
−−types−array=UA_TYPES_DI
−−types−array=UA_TYPES
−−existing deps/ua−nodeset/Schema/Opc.Ua.NodeSet2.xml
−−existing deps/ua−nodeset/DI/Opc.Ua.Di.NodeSet2.xml
−−xml deps/ua−nodeset/PLCopen/Opc.Ua.Plc.NodeSet2.xml
../opcua−open62541−server/src/nodeset/ua_namespace_plc
```

Now that we have all the upper Information model layers, we can compile the xml file generated by the MatIEC - representing the Information Model of the IEC 61131-3 program created in Beremiz - into it's equivalent C format. First copy the OPC UA xml mapped model file to examples/nodeset/.

```
cp matiec/tests/test.xml examples/nodeset/
```

After that, compile the mapped model by linking it with the three Information Models it extends - OPC UA, OPC UA DI and OPC UA IEC 61131-3.

```
python tools/nodeset_compiler/nodeset_compiler.py
--types-array=UA_TYPES
--existing deps/ua-nodeset/Schema/Opc.Ua.NodeSet2.xml
--existing deps/ua-nodeset/DI/Opc.Ua.Di.NodeSet2.xml
--existing deps/ua-nodeset/PLCopen/Opc.Ua.Plc.NodeSet2.xml
--xml examples/nodeset/test.xml
../opcua-open62541-server/src/nodeset/test
```

7.5 OPC UA Server

In the end, we'll have an header and a source file for each XML Information Model compiled. For each source and header pair, there is a main function that will add all the nodes specified in that information model to the server. This function takes as arguments the UA_Server and the Namespace variables. The functions must be run in the same order as the hierarchy of the information models: first the father of all fathers, the function from the Basic Information Model, than the DI, after that the IEC 61131-3 and finally the MatIEC mapped model. To test how the XML NodeSet Compiler and the open62541 OPC UA server can work together check my opcua-open62541-server bitbucket repository [83], it has a docker file you can build to put a OPC UA Server running with the Information Models created by the XML NodeSet Compiler. The C code used to build the open62541 OPC UA server with the included information models is shown in the listing 7.5 OPC UA Server

```
#include <signal.h>
#include <open62541.h>
// Include the headers generated by the XML NodeSet Compiler
#include <nodeset/ua_namespace_di.h>
#include <nodeset/ua_namespace_plc.h>
#include <nodeset/ua_types_di_generated.h>
#include <nodeset/ua_namespace_mapped.h>
```

```
UA_Boolean running = true;
static void stopHandler(int sig) {
    UA_LOG_INFO(UA_Log_Stdout,
                UA_LOGCATEGORY_USERLAND,
                "receive ctrl−C");
    running = false;
}

int main(void) {
    // SIGINT: interrupt signal such as ctrl−C
    signal(SIGINT, stopHandler);
    // SIGTERM: termination request
    signal(SIGTERM, stopHandler);

    UA_ServerConfig *config = UA_ServerConfig_new_default();
    UA_Server *server = UA_Server_new(config);

    UA_StatusCode retval;

    // Add the OPC UA DI Information Model namespace
    retval |= ua_namespace_di(server);
    if (retval != UA_STATUSCODE_GOOD) {
        UA_LOG_ERROR(UA_Log_Stdout,
                     UA_LOGCATEGORY_SERVER,
                     "Adding the DI namespace failed!");
        UA_Server_delete(server);
        UA_ServerConfig_delete(config);
        return (int) UA_STATUSCODE_BADUNEXPECTEDERROR;
    }

    // Add the OPC UA IEC 61131−3 Information Model namespace
    retval = ua_namespace_plc(server);
    if (retval != UA_STATUSCODE_GOOD) {
        UA_LOG_ERROR(UA_Log_Stdout,
                     UA_LOGCATEGORY_SERVER,
                     "Adding the PLC namespace failed!");
        UA_Server_delete(server);
        UA_ServerConfig_delete(config);
        return (int) UA_STATUSCODE_BADUNEXPECTEDERROR;
    }

    // Add the OPC UA MatIEC mapped namespace
    retval |= ua_namespace_mapped(server);
    if (retval != UA_STATUSCODE_GOOD) {
```

```
        UA_LOG_ERROR(UA_Log_Stdout,
                     UA_LOGCATEGORY_SERVER,
                     "Adding the mapped namespace failed!");
        UA_Server_delete(server);
        UA_ServerConfig_delete(config);
        return (int) UA_STATUSCODE_BADUNEXPECTEDERROR;
    }

    retval = UA_Server_run(server, &running);

    UA_Server_delete(server);
    UA_ServerConfig_delete(config);
    return (int) retval;
}
```

Listing 7.1: Example open62541 server code

7.6 OPC UA Client

The two best free OPC UA Clients found were the FreeOpcUa Client - an open source C++ and Python OPC UA Libraries - available in FreeOpcUa repository, and the UAExpert OPC UA Client - an OPC UA Client developed by Unified Automation - available in Unified Automation website.

The UAExpert Client was considered the best of this two options and was the one used to navigate the OPC UA Server *AddressSpace* during the debug process.

Chapter 8

Conclusion

The objective of providing support for OPC UA in the Beremiz softPLC was significantly accomplished. In the process two Architectures and two Code Generation modules for the MatIEC compiler were developed. The First Architectures relies on the XML NodeSet Compiler, an external tool, that adds a step to the compilation process.

In the Second Architecture designed, with the second code generation module for MatIEC developed, the OPC UA support is delivered with just the MatIEC Compiler, and no external software tool is needed.

The logic mapping of IEC 61131-3 into OPC UA Nodes - following the OPC UA Information Model for IEC 61131-3 - used in both MatIEC modules, maps the *CtrlConfigurations*, *CtrlResources*, *CtrlTasks*, *CtrlPrograms*, *CtrlFunctionBlocks*, *CtrlVariables* and it's respective *ObjectTypes*. It implements the *Mapping of elementary data types*, specified in 5.2.1. The Node structure implements the Mandatory *DeviceSet as entry point for enginnering applications*, specified in 6.1 and the *CtrlTypes Folder for server specific ObjectTypes* specified in 6.2. Work still in development includes *Mapping of derived data types* - 5.2.3, *Access Level* - 5.3.2 and *IEC CtrlVariable keywords* - 5.4.1.

With the XML output - following the OPC UA Information Model XML Schema - used in the first module the IEC 61131-3 program mapped to OPC UA is serialized into a format that can be read by a computer program - OPC UA Part 6 F.1.

The open62541 XML NodeSet Compiler builds on this capability and translates the XML into the open62541 C code implementing the UA Binary Schema. This provides UA Binary Message transfer between the Beremiz open62541 OPC UA Server used and OPC UA Clients.

8.1 Future Work

Future work include code optimization and re-factoring of the MatIEC OPC UA Generator code developed. More extensive testing can be done. The connection between the OPC UA Node Variables and the Beremiz softPLC variables still needs to be done using the open62541 callback

methods. The automatic generation of the OPC UA Server by the MatIEC is still in development. OPC UA specifications and Collaboration specifications are constantly being upgraded and new versions published. With the release of new versions the MatIEC modules need to be upgraded accordingly. The open62541 OPC UA implementation, is also constantly being upgraded, for compatibility purposes, if there's new releases of this software package, the OPC UA Server should also be upgraded.

References

[1] Thenextweb.com. Emerging markets are the growth generator of the digital world. URL: https://thenextweb.com/africa/2016/02/15/emerging-markets-are-the-growth-generator-of-the-digital-world/.

[2] Prof. Dr. H. Kirrmann. Industrial Automation. *EPFL,ABB Research Center, Baden, Switzerland*, 2005.

[3] OPC Foundation. History. URL: https://opcfoundation.org/about/opc-foundation/history/ [last accessed 2018-05-27].

[4] OPC Foundation. Interoperability for Industrie 4.0 and the Internet of Things. (November):1–48, 2017.

[5] OPC Foundation. OPC UA Part 3 - Address Space Model. *OPC Unified Architecture Specification*, Part 3, 2017.

[6] National Instruments. Why OPC UA Matters. pages 2017–2019, 2018.

[7] Karl Heinz John and Michael Tiegelkamp. IEC 61131-3: Programming industrial automation systems: Concepts and programming languages, requirements for programming systems, decision-making aids. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*, pages 1–390, 2010. doi:10.1007/978-3-642-12015-2.

[8] Andreas Otto and Klas Hellmann. IEC 61131: A general overview and emerging trends. *IEEE Industrial Electronics Magazine*, 3(4):27–31, 2009. doi:10.1109/MIE.2009.934793.

[9] Tutorialspoint.com. Compiler Design - Architecture. URL: https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm [last accessed 2018-06-03].

[10] Hacker Noon. Compilers and Interpreters. URL: https://hackernoon.com/compilers-and-interpreters-3e354a2e41cf [last accessed 2018-06-03].

[11] OPC Foundation and PLCopen. OPC UA Information Model for IEC 61131-3 - Release 1.00. 2010.

[12] OPC Foundation. OPC UA for Devices. *OPC Unified Architecture Companion Specification*, (Part DI: Devices Companion Specification), 2009.

[13] OPC Foundation. OPC UA Part 5 - Information Model. *OPC Unified Architecture Specification*, Part 5, 2017.

[14] International Standard IEC. IEC 61131-3. 2003, 1999. doi:10.1109/IEEESTD.2007.4288250.

[15] OPC Foundation. UA-Nodeset/Schema github repository. URL: https://github.com/OPCFoundation/UA-Nodeset/tree/master/Schema.

[16] OPC Foundation. UA-Nodeset/DI github repository. URL: https://github.com/OPCFoundation/UA-Nodeset/tree/master/DI.

[17] OPC Foundation. UA-Nodeset/PLCopen github repository. URL: https://github.com/OPCFoundation/UA-Nodeset/tree/master/PLCopen.

[18] Klaus (World Economic Forum) Schwab. *The Fourth Industrial Revolution*. 2016. arXiv:arXiv:1011.1669v3, doi:10.1017/CBO9781107415324.004.

[19] OPCconnect. History of OPC. URL: http://www.opcconnect.com/history.php [last accessed 2018-05-27].

[20] M. H. Schwarz and J. Borcsok. A survey on OPC and OPC-UA: About the standard, developments and investigations. *2013 24th International Conference on Information, Communication and Automation Technologies, ICAT 2013*, 2013. doi:10.1109/ICAT.2013.6684065.

[21] Y. Shimanuki. OLE for process control (OPC) for new industrial automation systems. *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028)*, 6:1048–1050, 1999. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=816721, doi:10.1109/ICSMC.1999.816721.

[22] Li Zheng Li Zheng and H. Nakagawa. OPC (OLE for process control) specification and its developments. *Proceedings of the 41st SICE Annual Conference. SICE 2002.*, 2:917–920, 2002. doi:10.1109/SICE.2002.1195286.

[23] MatrikonOPC.com. OPC Data Access (OPC DA) Versions & Compatibility. URL: https://www.matrikonopc.com/opc-server/opc-data-access-versions.aspx [last accessed 2018-05-23].

[24] OPC Foundation. What is OPC? URL: https://opcfoundation.org/about/what-is-opc/ [last accessed 2018-05-23].

[25] Matlab. OPC Data: Value, Quality, and TimeStamp. URL: https://www.mathworks.com/help/opc/ug/understanding-opc-data-value-quality-and-timestamp.html [last accessed 2018-05-28].

[26] OPC Foundation. What is OPC Classic? URL: https://opcfoundation.org/faq/what-is-opc-classic/ [last accessed 2018-05-23].

[27] Mai Son and Myeong Jae Yi. A study on OPC specifications: Perspective and challenges. *2010 International Forum on Strategic Technology, IFOST 2010*, pages 193–197, 2010. doi:10.1109/IFOST.2010.5668110.

[28] OPC Foundation. What is OPC UA? URL: https://opcfoundation.org/faq/what-is-opc-ua/ [last accessed 2018-05-23].

[29] OPC Foundation. OPC UA Part 1 - Overview and Concepts. *OPC Unified Architecture Specification*, Part 1, 2017.

[30] Tom Hannelius, Mikko Salmenperä, and Seppo Kuikka. Roadmap to adopting OPC UA. *IEEE International Conference on Industrial Informatics (INDIN)*, pages 756–761, 2008. doi:10.1109/INDIN.2008.4618203.

[31] Plattform Industrie 4.0. What is Industrie 4.0? URL: https://www.plattform-i40.de/I40/Navigation/EN/Industrie40/WhatIsIndustrie40/what-is-industrie40.html [last accessed 2018-06-22].

[32] Andreja Rojko. Industry 4.0 Concept: Background and Overview. *International Journal of Interactive Mobile Technologies (iJIM)*, 11(5):77, 2017. URL: http://online-journals.org/index.php/i-jim/article/view/7072, doi:10.3991/ijim.v11i5.7072.

[33] Plattform Industrie 4.0. Plattform Industrie 4.0. URL: https://www.plattform-i40.de/I40/Navigation/EN/ThePlatform/PlattformIndustrie40/plattform-industrie-40.html [last accessed 2018-06-22].

[34] Michael Hoffmeister. Industrie 4.0: The Industrie 4.0 Component. 1.0(April):2, 2015. URL: http://www.zvei.org/Downloads/Automation/ZVEI-Industrie-40-Component-English.pdf.

[35] VID/VDE. Reference Architecture Model Industrie 4.0 (RAMI4.0). *Igarss 2014*, 0(1):28, 2015. arXiv:arXiv:1011.1669v3, doi:10.1007/s13398-014-0173-7.2.

[36] Plattform Industrie 4.0. Robot Revolution Initiative. URL: https://www.plattform-i40.de/I40/Redaktion/EN/Standardartikel/international-cooperation-rri.html [last accessed 2018-06-22].

[37] U.S. Chamber of Commerce. Made in China 2025: Global Ambitions Built on Local Protections. page 84, 2017. URL: https://www.uschamber.com/sites/default/files/final{_}made{_}in{_}china{_}2025{_}report{_}full.pdf.

[38] European Commission. France : Industrie du Futur. (January), 2017. URL: https://webgate.acceptance.ec.europa.eu/growth/tools-databases/dem/monitor/content/france-industrie-du-futur.

[39] Governo Italiano. Industria 4.0. URL: http://www.sviluppoeconomico.gov.it/index.php/it/industria40 [last accessed 2018-06-22].

[40] Plattform Industrie 4.0 and Industrial Internet Consortium. Architecture Alignment and Interoperability An Industrial Internet Consortium and Plattform Industrie 4.0 Joint Whitepaper. page 19, 2018. URL: http://www.iiconsortium.org/pdf/JTG2{_}Whitepaper{_}final{_}20171205.pdf, doi:IIC:WHT:IN3:V1.0:PB:20171205.

[41] European Commission. National initiatives. URL: https://ec.europa.eu/growth/tools-databases/dem/monitor/category/national-initiatives [last accessed 2018-06-22].

[42] European Commission. Multi-stakeholder platform on SDGs. URL: https://ec.europa.eu/info/strategy/international-strategies/global-topics/sustainable-development-goals/multi-stakeholder-platform-sdgs{_}en [last accessed 2018-06-22].

[43] European Commission. industry 4.0. URL: https://ec.europa.eu/growth/tools-databases/dem/monitor/tags/industry-40 [last accessed 2018-06-22].

[44] Demetrius Klitou, Johannes Conrads, and Morten Rasmussen. Key lessons from national industry 4.0 policy initiatives in Europe. *Digital Transformation Monitor*, (May), 2017. URL: https://ec.europa.eu/growth/tools-databases/dem.

[45] Martin Hankel and Bosch Rexroth. The Reference Architectural Model Industrie 4.0 (RAMI 4.0). *ZWEI: Die Elektroindustrie*, 1(April):1–2, 2015.

[46] Plattform Industrie 4.0. Reference Architectural Model Industrie 4.0 (RAMI 4.0) - An Introduction. 0:21, 2016. URL: https://www.plattform-i40.de/I40/Redaktion/EN/Downloads/Publikation/rami40-an-introduction.pdf?{_}{_}blob=publicationFile{&}v=7.

[47] Shi-Wan (Thingswise/Intel) Lin, Bradford (GE) Miller, Jacques (Fujitsu) Durand, Graham (IBM) Bleakley, Amine (GE) Chigani, Robert (MITRE) Martin, Brett (RTI) Murphy, and Mark Crawford (SAP). The Industrial Internet of Things Volume G1: Reference Architecture. *Industrial Internet Consortium*, 1.80(November):1 – 7, 2017. URL: http://www.iiconsortium.org/IIC{_}PUB{_}G1{_}V1.80{_}2017-01-31.pdf{%}0Ahttp://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/intelligent-process-automation-the-engine-at-the-core-of-the-next-generation-ope//www.mckinsey.com/i.

[48] Automation World. Industrial Internet Consortium and Plattform Industrie 4.0 Align Architectures. URL: https://www.automationworld.com/industrial-internet-consortium-and-plattform-industrie-40-align-architectures [last accessed 2018-06-22].

[49] German Federal Ministry for Economic Affairs and Energy and Standardization Administration of the P.R.C. Alignment Report for Reference Architectural Model for Industrie 4.0/ Intelligent Manufacturing System Architecture - Sino-German Industrie 4.0/Intelligent Manufacturing Standardisation Sub-Working Group. page 36, 2018. URL: https://www.plattform-i40.de/I40/Redaktion/EN/Downloads/Publikation/hm-2018-manufactoring.pdf?{_}{_}blob=publicationFile{&}v=2.

[50] Joint Working Groups. Alliance Industrie du Futur. (April 2016), 2017.

[51] Plattform Industrie 4.0. International Cooperation. URL: https://www.plattform-i40.de/I40/Navigation/EN/InPractice/International/international.html [last accessed 2018-06-22].

[52] Stefan Hoppe, Beckhoff Automation, Development Goals, Vojna Ngjeqari, Daymon Thompson, Twincat Product Specialist, Beckhoff Automation, Ming-zhou Liu, Cong-hu Liu, Maogen Ge, Yuan Zhang, Qing-hua Zhu, Veronika Stumpf, Hiroshi Sasaki, Facts For, Decision Makers, Thomas J Burke, Position Paper, Liu Yang Caict, Urban Development, German International, Development Cooperation, Joe Ottenhof, Industrie Consulting, Henning Kagermann, Reiner Anderl, International Benchmark, Gordon Moore, The Internet Protocol, Working Group, National Academy, Working Group, Federal Ministry, Cfk-valley Stade Convention, Cyber-physical Systems, Directorate General, F O R Internal, Francisco Almada-lobo, T. Hermann, M.; Pentek, and Helbig (Deutsche Post Ag). Henning, Kagermann(National Academy of Science and Engineering). Wolfgang, Wahlster (German Research Center for Artificial Intelligence). Johannes. OPC UA as a Bridge Between IT and Automation. *Final report of the Industrie 4.0 WG*, 4(March):82, 2015. arXiv:1011.1669v3, doi:10.1109/HICSS.2016.488.

[53] Stefan Hoppe. There is no Industrie 4.0 without OPC UA. *OPC Foundation*, pages 0–1, 2017. URL: https://www.pc-control.net/pdf/012017/technology/pcc{_}0117{_}industrie-40-opc-ua{_}e.pdf.

[54] Germann Electrical Association and Electronica Manufacturers'. What Criteria do Industrie 4 . 0 Products Need to Fulfil ? (April), 2017. URL: https://www.zvei.org/en/press-media/publications/what-criteria-do-industrie-40-products-need-to-fulfil/.

[55] VDMA - Verband Deutscher Maschinen- und Anlagenbau. Industrie 4.0 Communication Guideline. URL: https://industrie40.vdma.org/en/viewer/-/v2article/render/20625194.

[56] Automation World. Industrial Internet Consortium and Plattform Industrie 4.0 Align Architectures. URL: https://www.automationworld.com/industrial-internet-consortium-and-plattform-industrie-40-align-architecture [last accessed 2018-06-22].

[57] OPC Connect. OPC UA in South Korea and Hannover Messe, Germany – OPC Connect. URL: https://opcconnect.opcfoundation.org/2017/04/opc-ua-in-south-korea-and-hannover-messe-germany/ [last accessed 2018-06-23].

[58] OPC Foundation. Markets and Collaboration. URL: https://opcfoundation.org/markets-collaboration/ [last accessed 2018-07-03].

[59] Library.automationdirect.com. History of the PLC | Library.AutomationDirect.com | #1 Value. URL: https://library.automationdirect.com/history-of-the-plc/ [last accessed 2018-05-30].

[60] Barn.org. The History of the PLC. URL: http://www.barn.org/FILES/historyofplc.html [last accessed 2018-05-30].

[61] Control Design. A very short history of PLC programming platforms. URL: https://www.controldesign.com/articles/2017/a-very-short-history-of-plc-programming-platforms/ [last accessed 2018-05-30].

[62] Function Block Diagram and Ladder Diagram. Understanding the IEC61131-3 Programming Languages. *Control Engineering*, page 6, 2009.

[63] Edouard Tisserant, Laurent Bessard, and Mário De Sousa. An Open Source IEC 61131-3 Integrated Development Environment. 33(0), 2007.

[64] PLCopen. IEC 61131 Standards. URL: http://www.plcopen.org/pages/tc1{_}standards/ [last accessed 2018-05-31].

[65] PLCopen. Advantages for Users IEC 61131. URL: http://www.plcopen.org/pages/benefits/benefits{_}for{_}users{_}iec61131/ [last accessed 2018-05-31].

[66] Beremiz.org. Beremiz | About. URL: https://beremiz.org/ [last accessed 2018-03-17].

[67] PLCopen. XML intro. URL: http://www.plcopen.org/pages/tc6{_}xml/xml{_}intro/index.htm [last accessed 2018-06-01].

[68] Codecademy. MVC: Model, View, Controller. URL: https://www.codecademy.com/articles/mvc [last accessed 2018-06-14].

[69] Medium.com. Back-Propagation is very simple. Who made it Complicated ? URL: https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c [last accessed 2018-06-14].

[70] CANFestival.org. The CanFestival CANopen stack manual. pages 1–42.

[71] National Instruments. The Basics of CANopen. URL: http://www.ni.com/white-paper/14162/en/ [last accessed 2018-03-17].

[72] CAN in Automation (CiA). CAN knowledge. URL: https://www.can-cia.org/can-knowledge/ [last accessed 2018-03-17].

[73] Philippe Grosjean. SciViews GUI API. pages 1–8, 2015. URL: http://www.sciviews.org/SciViews-R.

[74] Info.kepware.com. Four Reasons Why No Standard is the King of Automation. URL: https://info.kepware.com/blog/four-reasons-why-no-standard-is-the-king-of-automation [last accessed 2018-06-23].

[75] Siemens. Building automation and control systems. 2016.

[76] Mário De Sousa. MatPLC - The truly open automation controller. *IECON Proceedings (Industrial Electronics Conference)*, 3:2278–2283, 2002. doi:10.1109/IECON.2002.1185327.

[77] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques & Tools*. 2007.

[78] John Levine. *Flex & Bison, Unix Text Processing Tools*. O'Reilly Media, 2009.

[79] GNU Project - Free Software Foundation. Bison. URL: https://www.gnu.org/software/bison/ [last accessed 2018-06-04].

[80] OPC Foundation. UA-Nodeset github repository. URL: https://github.com/OPCFoundation/UA-Nodeset.

[81] open62541.org. open62541. URL: https://open62541.org/.

[82] OPC Foundation. OPC UA Part 6 - Mappings. *OPC Unified Architecture Specification*, Part 6:5, 2017.

[83] ee12099. open62541 OPC UA Server bitbucket repository. URL: https://bitbucket.org/ee12099/opcua-open62541-server/src/master/.